Lecture Notes in Computer Science 5374

Ponnuswamy Sadayappan Manish Parashar
Ramamurthy Badrinath Viktor K. Prasanna (Eds.)

High Performance Computing - HiPC 2008

15th International Conference
Bangalore, India, December 17-20, 2008
Proceedings

Springer

Volume Editors

Ponnuswamy Sadayappan
The Ohio State University, Department of Computer Science and Engineering
2015 Neil Avenue, Columbus, OH 43210, USA
E-mail: sadayappan.1@osu.edu

Manish Parashar
Rutgers, The State University of New Jersey
Department of Electrical and Computer Engineering
94 Brett Road , Piscataway , NJ 08854, USA,
E-mail: parashar@caip.rutgers.edu

Ramamurthy Badrinath
Hewlett-Packard ISO, Sy 192
Whitefield Road, Mahadevapura Post, Bangalore 560048, India
E-mail: ramamurthy.badrinath@hp.com

Viktor K. Prasanna
University of Southern California, Department of Electrical Engineering
Los Angeles, CA 90089-2562, USA
E-mail: prasanna@usc.edu

Library of Congress Control Number: 2008940427

CR Subject Classification (1998): B.2.4, D.1, C.1, F.1, G.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-89893-X Springer Berlin Heidelberg New York
ISBN-13 978-3-540-89893-1 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12577897 06/3180 5 4 3 2 1 0

Message from the Program Chair

Welcome to the 15th International Conference on High Performance Computing (HiPC 2008), held at Bangaluru (formerly called Bangalore), India. Features at this year's conference included a single-track program with 46 technical papers selected from 317 submissions, four keynote talks by renowned experts from academia, two tutorials from industry, five workshops, a student research symposium, and a user/industry symposium.

As in recent years, a large number of technical papers were submitted for consideration, covering six topic areas of focus: algorithms, applications, architecture, communication networks, mobile/sensor computing, and systems software. In reviewing the 317 submissions, we used a two-phase approach as done over the last few years with HiPC. A first screening pass was made through all papers by the area Vice Chairs to identify papers that were clearly out of scope for the conference, either because the topic did not fit the conference or the paper was of a survey nature, without original research content. After the first pass, the remaining 285 submissions were reviewed, seeking at least three reviews from Program Committee members. A number of external reviewers also provided reviews for papers. A total of 978 reviews were obtained for the papers. For papers with significantly differing opinions among Program Committee members, e-mail discussions were used to find consensus. Finally 46 papers were selected for presentation in 8 sessions.

One outstanding paper was selected for the best paper award. Each area Vice Chair first nominated one or two of the best papers from their area. These papers and their reviews were studied by the Vice Chairs, who then made their recommendations for the best paper award. Based on the recommendations from the Vice Chairs, the paper "Scalable Multi-cores with Improved Per-core Performance Using Off-the-Critical Path Reconfigurable Hardware," authored by Tameesh Suri and Aneesh Aggarwal, was selected for the best paper award.

This year's program featured keynote presentations from four distinguished speakers: Wolfgang Gentzsch on the European distributed supercomputing infrastructure, David Peleg on networked computers, Mary Wheeler on a computational environment for subsurface modeling, and Laxmikant Kale on the "the excitement in parallel computing."

The quality of the technical program is critically dependent on the efforts of the Program Committee members in providing reviews for the submitted papers. I thank the 103 members of this year's Program Committee. The assignment of papers to members of the Program Committee was managed by the area Vice Chairs. I was extremely fortunate to work with dedicated Vice Chairs. The Vice Chairs were David Bader (Algorithms), Alan Sussman (Applications), David Kaeli and Martin Schulz (Architecture), José de Souza (Communication Networks), Chen-Khong Tham (Mobile and Sensor Computing), and Cho-Li Wang (Systems Software). I am very grateful to them for their hard work in handling the reviewing within their areas and their significant contributions at the distributed virtual Program Committee meeting. Each paper's review recommendations were carefully checked for consistency; in many instances, the Vice Chairs read the papers themselves when the reviews did not seem sufficient to make a decision.

Throughout the reviewing process, I received a tremendous amount of help and advice from General Co-chair Manish Parashar, Steering Chair Viktor Prasanna, and last year's Program Chair Srinivas Aluru; I am very grateful to them. My thanks also go to the Publications Chair Sushil Prasad for his outstanding efforts in putting the proceedings together. Finally, I thank all the authors for their contributions to a high-quality technical program. I wish all the attendees a very enjoyable and informative meeting.

December 2008 P. Sadayappan

Message from the General Co-chairs and the Vice General Co-chairs

On behalf of the organizers of the 15th International Conference on High-Performance Computing (HiPC), it is our pleasure to present these proceedings and we hope you will find them exciting and rewarding.

The HiPC call for papers, once again, received an overwhelming response, attracting 317 submissions from 27 countries. P. Sadayappan, the Program Chair, and the Program Committee worked with remarkable dedication to put together an outstanding technical program consisting of the 46 papers that appear in these proceedings.

Several events, complementing this strong technical program, made HiPC 2008 a special and exciting meeting. As in previous years, the HiPC 2008 keynotes were presented by internationally renowned researchers. HiPC 2008 also featured a full-day student research symposium prior to the main conference, which consisted of presentations and posters by students highlighting their research. The conference once again had an industry and user symposium focused on "High-Performance Computing Technologies, Applications and Experience," which ran in parallel to the main track, to bring together the users and providers of HPC. This symposium and the main conference came together for a special plenary panel on "Cloud Computing." There were several industry and research exhibits complementing the symposium and the conference main track. The meeting was preceded by a set of tutorials and workshops highlighting new and emerging aspects of the field.

Arranging an exciting meeting with a high-quality technical program is easy when one is working with an excellent and dedicated team and can build on the practices and levels of excellence established by a quality research community. HiPC 2008 would not have been possible without the tremendous efforts of the many volunteers. We would like to acknowledge the critical contributions of each one.

We would like to thank P. Sadayappan, Program Chair, and the Program Committee for their efforts in assembling such an excellent program, and the authors who submitted the high-quality manuscripts from which that program was selected. We would also like to thank the presenters of the keynotes, posters and tutorials, the organizers of the workshops, and all the participants, who completed the program.

We would specially like to thank Viktor Prasanna, Chair of the HiPC Steering Committee, for his leadership, sage guidance, and untiring dedication, which have been key to the continued success of the conference. We would also like to welcome our new volunteers to the team - your efforts are critical to the continued success of this conference. Finally, we would like to gratefully acknowledge our academic and industry sponsors including IEEE Computer Society, ACM SIGARCH, Infosys, DELL, NetApp, Intel, HP, IBM, Yahoo!, Cray and Mellanox.

December 2008

Manish Parashar
Ramamurthy Badrinath
Rajendra V. Boppana
Rajeev Muralidhar

Message from the Steering Chair

It is my pleasure to welcome you to the proceedings of the 15th International Conference on High Performance Computing and to Bengaluru, the leading center of IT activity in India.

My "thank you" goes to many volunteers whose dedicated effort over the past year has made this conference a successful endeavor. P. Sadayappan, our Program Chair, has done an outstanding job in putting together an excellent technical program. I am indebted to him for his thorough evaluation of the submitted manuscripts and his relentless efforts to further improve the quality of the technical program. Manish Parashar and Ramamurthy Badrinath as General Co-chairs provided the leadership in resolving numerous meeting-related issues and putting together the overall program including the workshops and tutorials. They were ably assisted by Rajeev Muralidhar, Vice General Co-chair. The industry track was coordinated by Rama Govindaraju with assistance from Frank Baetke and Santosh Sreenivasan. We have several continuing as well as new workshops. These workshops were coordinated by Manimaran Govindarasu. The website was maintained by Yinglong Xia. Animesh Pathak acted as the Production Chair overseeing various activities related to the Web and creating publicity materials. The Student Research Symposium was organized by Ashok Srinivasan and Rajeev Thakur. Rajeev Sivaram assisted us with the tutorials. The local arrangements were handled by C. Kalyana Krishna and Raghavendra Buddi. Sushil Prasad interfaced with the authors and Springer to bring out the proceedings. Manisha Gajbe and Ashok Srinivasan handled the publicity for us. Sally Jelinek and Jyothsna Kasthurirengan acted as the Registration Co-chairs. Ajay Gupta and Thondiyil Venugopalan were in charge of the meeting finances. Sumam David and Madhusudan Govindaraju administered the student scholarships.

I would like to thank all our volunteers for their tireless efforts. The meeting would not be possible without the enthusiastic effort and commitment of these individuals.

Major financial support for the meeting was provided by several leading IT companies and multinationals operating in India. I would like to acknowledge the following individuals and their organizations for their support:

- N.R. Narayana Murthy, Infosys
- Kris Gopalakrishnan, Infosys
- David Ford, NetApp
- Siddhartha Nandi, NetApp
- B. Rudramuni, Dell India
- Ramesh Rajagopalan, Dell India
- Reza Rooholamini, Dell
- V. Sridhar, Satyam
- Prabhakar Raghavan, Yahoo! Inc.
- Arun Ramanujapuram, Yahoo! India R&D
- Vittal Kini, Intel Research, India

- Biswadeep Chatterjee, Intel Research, India
- Venkat Natarajan, Intel Research, India
- Manish Gupta, IBM India
- Dinkar Sitaram, HP India
- Faisal Paul, HP India
- Venkat Ramana, Hinditron Infosystems

December 2008 Viktor K. Prasanna

Conference Organization

General Co-chairs

Manish Parashar	Rutgers University, USA
Ramamurthy Badrinath	HP, India

Vice General Co-chairs

Rajeev Muralidhar	Intel, India
Rajendra V. Boppana	University of Texas at San Antonio, USA

Program Chair

P. Sadayappan	Ohio State University, USA

Program Vice Chairs

Algorithms

David Bader	Georgia Institute of Technology, USA

Applications

Alan Sussman	University of Maryland, USA

Architecture

David Kaeli	Northeastern University, USA
Martin Schulz	Lawrence Livermore Laboratory, USA

Communication Networks

José Neuman de Souza	Federal University of Ceará, Brazil

Mobile and Sensor Computing

Chen-K Tham	National University of Singapore

Systems Software

Cho-li Wang	Hong Kong University

Steering Chair

Viktor K. Prasanna	University of Southern California, USA

Workshops Chair

Manimaran Govindarasu	Iowa State University, USA

Student Research Symposium Co-chairs

Ashok Srinivasan	Florida State University, USA
Rajeev Thakur	Argonne National Laboratory, USA

Tutorials Chair

Rajeev Sivaram	Google, USA

Industry Liaison Chair

Rama K. Govindaraju	Google, USA

HiPC User Symposium Co-chairs

Rama K. Govindaraju	Google, USA
Santosh Sreenivasan	Talentain, India

Cyber Co-chairs

Ananth Narayan	Intel, India
Kamesh Madduri	Georgia Institute for Technology, USA
Yinglong Xia	University of Southern California, USA

Finance Co-chairs

Ajay Gupta	Western Michigan University, USA
Thondiyil Venugopalan	India

Local Coordination Co-chairs

Kalyana Krishna	Talentain, India
Raghavendra Buddi	NetApp, India

Production Chair

Animesh Pathak	University of Southern California, USA

Publications Chair

Sushil K. Prasad	Georgia State University, USA

Publicity Co-chairs

Ashok Srinivasan	Florida State University, USA
Manisha Gajbe	Georgia Institute of Technology, USA

Registration Co-chairs

Sally Jelinek	Electronics Design Associates, USA
Jyothsna Kasthurirengan	Fiberlink, India

Scholarships Co-chairs

Sumam David	NITK, India
Madhusudhan Govindaraju	SUNY Binghamton, USA

Steering Committee

Steering Committee 2008 membership also includes the General Co-chairs, Program Chairs and Vice General Co-chairs from 2007 and 2008.

P. Anandan	Microsoft Research, India
David A. Bader	Georgia Institute of Technology, USA
Ramamurthy Badrinath	HP, India
Rudramuni B.	Dell India R&D, Bangalore, India
Frank Baetke	HP, USA
Anirban Chakrabarti	Infosys, India
R. Govindarajan	Indian Institute of Science, India
Harish Grama	IBM, India
Manish Gupta	India Systems and Technology Lab, IBM, India
Vittal Kini	Intel, India
Ashwini Nanda	Computational Research Lab., India
Viktor K. Prasanna	University of Southern California, USA (Chair)
Venkat Ramana	Cray-Hinditron, India
Sartaj Sahni	University of Florida, USA
Santosh Sreenivasan	Talentain Technologies, India
V. Sridhar	Satyam Computer Services Ltd., India
Harrick M. Vin	Tata Research, Development & Design Center, Pune, India

Program Committee

Algorithms

Srinivas Aluru	Iowa State University, USA
Guojing Cong	IBM T.J. Watson Research Center, USA
Camil Demetrescu	University of Rome, Italy

Devdatt Dubhashi	Chalmers University of Technology, Sweden
Matteo Frigo	Cilk Arts, USA
Anshul Gupta	IBM T.J. Watson Research Center, USA
Klaus Jansen	University of Kiel, Germany
Jeremy Johnson	Drexel University, USA
Christos Kaklamanis	University of Patras, Greece
Ananth Kalyanaraman	Washington State University, USA
Jesper Larsson Traeff	NEC Laboratories, Europe
Kamesh Madduri	Georgia Institute of Technology, USA
Greg Malewicz	Google, USA
Madhav Marathe	Virginia Tech., USA
Geppino Pucci	University of Padova, Italy
Sandeep Sen	Indian Institute of Technology, Delhi, India
Christian Sohler	University of Paderborn, Germany
Philippas Tsigas	Chalmers University of Technology, Sweden
Tiffani Williams	Texas A & M University, USA

Applications

Henrique Andrade	IBM T.J. Watson Research Center, USA
Umit Catalyurek	Ohio State University, USA
Nikos Chrisochoides	College of William and Mary, USA
I-Hsin Chung	IBM T.J. Watson Research Center, USA
Lois Curfman McInnes	Argonne National Laboratory, USA
Jens Gustedt	LORIA, France
Fumihiko Ino	Osaka University, Japan
P.J. Narayanan	Indian Institute of Information Technology, Hyderabad, India
Daniel S. Katz	Louisiana State University, USA
Jim Kohl	Oak Ridge National Laboratory, USA
Kalyan Kumaran	Argonne National Laboratory, USA
Gary Kumfert	Lawrence Livermore National Laboratory, USA
Steve Parker	University of Utah, USA
Padma Raghavan	Pennsylvania State University, USA
Ramesh Rajagopalan	Dell, India
Satish Vadhiyar	Indian Institute of Science, India
Yao Zheng	Zhejiang University, China

Architecture

Rosa Badia	Barcelona Supercomputer Center, Spain
Rajeev Balasubramonian	University of Utah, USA
Anasua Bhowmik	AMD, India
Antonio Gonzalez	Intel Barcelona Research Center, Spain
Ananth Grama	Purdue University, USA
Kim Hazelwood	University of Virginia, USA
Helen Karatza	Aristotle University of Thessaloniki, Greece

Sriram Krishnamoorthy	Pacific Northwest National Laboratory, USA
Hrishi Murukkathampoondi	AMD, India
Ranjani Parthasarathy	Anna University, India
Srinivasan Parthasarathy	Ohio State University, USA
Rodric Rabbah	IBM T.J. Watson Research Center, USA
J. Ramanujam	Louisiana State University, USA
Partha Ranganathan	HP Research Laboratories, USA
Pat Teller	University of Texas at El Paso, USA
Ramon Canal	Universitat Politécnica de Catalunya, Spain
Timothy Pinkston	University of Southern California, USA
Peter Varman	Rice University, USA
Youfeng Wu	Intel, USA

Communication Networks

Pavan Balaji	Argonne National Laboratory, USA
Rajendra Bopanna	University of Texas at San Antonio, USA
Ron Brightwell	Sandia National Laboratory, USA
Rajkumar Kettimuthu	Argonne National Laboratory, USA
John Kim	Northwestern University, USA
Jiuxing Liu	IBM T.J. Watson Research Center, USA
David Lowenthal	University of Georgia, USA
Amith Mamidala	Ohio State University, USA
Jarek Nieplocha	Pacific Northwest National Laboratory, USA
Thirumale Niranjan	NetApp Research, India
Fabrizio Petrini	IBM T.J. Watson Research Center, USA
Sayantan Sur	IBM T.J. Watson Research Center, USA
Vinod Tipparaju	Pacific Northwest National Laboratory, USA
Weikuan Yu	Oak Ridge National Laboratory, USA
Xin Yuan	Florida State University, USA

Mobile and Sensor Computing

Kevin Almeroth	University of California at Santa Barbara, USA
Twan Basten	Technical University of Eindhoven, The Netherlands
Mun-Choon Chan	National University of Singapore, Singapore
Eylem Ekici	Ohio State University, USA
Polly Huang	National Taiwan University, Taiwan
Vana Kalogeraki	University of California at Riverside, USA
Peng-Yong Kong	Institute for Infocomm Research, Singapore
Victor Leung	University of British Columbia, Canada
Jun Luo	University of Waterloo, Canada
Hoon-Tong Ngin	Infocomm Development Authority, Singapore
Marimuthu Palaniswami	University of Melbourne, Australia
Karim Seada	Nokia Research, USA
Winston Seah	Institute for Infocomm Research, Singapore

Vikram Srinivasan	Alcatel-Lucent Bell Labs, India
Wendong Xiao	Institute for Infocomm Research, Singapore
Athanasios Vasilakos	University of Western Macedonia, Greece

Systems Software

Rajkumar Buyya	University of Melbourne, Australia
Yeh-Ching Chung	National Tsing Hua University, Taiwan
Bronis De Supinski	Lawrence Livermore National Laboratory, USA
R. Govindarajan	Indian Institute of Science, India
Jaejin Lee	Seoul National University, Korea
Kuan-Ching Li	Providence University, Taiwan
Philippe Navaux	Federal University of Rio Grande do Sul, Brazil
Lawrence Rauchwerger	Texas A & M University, USA
Vijay A. Saraswat	IBM T.J. Watson Research Center, USA
Henk Sips	Delft University of Technology, The Netherlands
Ninghui Sun	Chinese Academy of Sciences, China
Yoshio Tanaka	National Inst. of Advanced Industrial Science and Technology, Japan
Putchong Uthayopas	Kasetsart University, Thailand
Richard Vuduc	Georgia Institute of Technology, USA
Laurence T. Yang	St. Francis Xavier University, Canada

Workshops

Workshop on Grid and Utility Computing

Organizers

Anirban Chakrabarti	Infosys Technologies, India
Shubhashis Sengupta	Oracle, India

Workshop on Service-Oriented Engineering and Optimizations

Organizers

Badrinath Ramamurthy	HP India
Geetha Manjunath	HP India

Workshop on Next-Generation Wireless Networks

Organizers

B. Prabakaran	University of Texas, Dallas, USA
S. Dharmaraja	Indian Institute of Technology, Delhi, India

High-Performance FPGA/Reconfigurable Computing

Organizers

Tirumale Ramesh	The Boeing Company, USA
Venkata Ramana	Hinditron-CRAY, India

Cooling of HiPC Systems: Why Should Computer Scientists Care?

Organizers

Venkat Natarajan (Chair)	Intel Corporation
Anand Deshpande (Co-chair)	Intel Corporation

Tutorials

High-Performance Computing with CUDA

Sanjiv Satoor	NVIDIA corp
Punit Kishore	NVIDIA corp

Hadoop – Delivering Petabyte-Scale Computing and Storage on Commodity Hardware

Yahoo Bangalore Cloud Computing Team	Yahoo, India

List of Reviewers

In addition to the PC members, the following colleagues provided reviews for HiPC 2008 papers. Their help is gratefully acknowledged.

Virat Agarwal
Gagan Agrawal
Rui Aguiar
Gheorge Almasi
Alexander van Amesfoort
Chee Ang
Paolo Bertasi
Vandy Berten
Elisa Bertino
Cristian Bertocco
Mauro Bianco
Karima Boudaoud
Steven Brandt
Francisco Brasileiro
Darius Buntinas
R.C. Hansdah
Jason C. Hung
Ken C.K. Tsang
Mark C.M. Tsang
Qiong Cai
Enrique Carrera
Joaquim Celestino
Marcia Cera
Girish Chandramohan
Chin-Chen Chang
Bin Chen
Jianxia Chen
Min Chen
Raphael Coeffic
Luis Costa
Pilu Crescenzi
Chirag Dave
Sheng Di
Jiun-Hung Ding
Evgueni Dodonov
Kshitij Doshi
Yew Fai Wong
Carlo Fantozzi
Paulo Fernandes
Marcial Fernandez
Andrei Formiga
Luca Foschini
Miguel Franklin de Castro
Mario Freire
Mrugesh Gajjar
Anilton Garcia
Yu Ge
Stephane Genaud
Amol Ghoting
Enric Gibert
William Giozza
Antonio Gomes
Danielo Gomes
Jose Gonzalez
Raju Gottumukkala
Manimaran Govindarasu
Helio Guardia
John Gunnels
Mingding Han
R. Hansdah
Ramin Hekmat
Kevin Ho
Roy Ho
Sai Ho Wong
Tuan Hoang
Meng-Ju Hsieh

Chin-Jung Hsu
Haoyu Hu
Kuo-Chan Huang
Jason Hung
Andrei Hutanu
Borhan Jalaeian
Shantenu Jha
Yingyin Jiang
Qing Jun Zhang
Francisco Junior
Carlos Kamienski
Rodrigo Kassick
Ian Kelley
Gaurav Khanna
Hyesoon Kim
Joohyun Kim
Kok Kiong Yap
David Koppelman
Ka Kui Ma
Archit Kulshrestha
Benjamin Lee
Arnaud Legrand
Tieyan Li
Wei Lih Lim
Meng-How Lim
Shih-Hsiang Lo
Tie Luo
Nicolas Maillard
Gianluca Maiolini
Konstantin Makarychev
R. Manikantan
Joberto Martins
Lorne Mason
Francisco Massetto
Collin McCurdy
Rodrigo Mello
Lucas Mello Schnorr
Luciano Mendes
Hashim Mohamed
Carlos Molina
Gregory Mounie
Y.N. Srikant
Ghasem Shirazi
Soumitra Nandy
Sivaramakrishnan
Narayanan
Lee-Ling Ong
Sreepathi Pai
Thomas Panas
Qixiang Pang
Ioannis Papadopoulos
Sriram Pemmaraju
Xiaoming Peng
Maikel Pennings
Angelo Perkusich
Marcelo Pias
Hwee Pink Tan
Dario Pompili
Monica Py
R. Vasudha
Kaushik Rajan
Easwaran Raman
Rodrigo Righi
Eduardo Rocha
Philip Roth
Lifeng Sang
Aldri Santos
Andre Santos
Abhinav Sarje
Daniele Scarpazza
Srinivasan Seetharaman
Aiyampalayam Shanthi
Jeonghee Shin
Chee Shin Yeo
Zheng Shou Kang
Francesco Silvestri
Gurmeet Singh
Prasun Sinha
Tim Smith
Wee-Seng Soh
Joo-Han Song
Francois Spies
K.N. Sridhar
Mukundan Sridharan
Y.N. Srikant
Michael Stanton
Wen Su
Xu Su
Dharmashankar
Subramanian
Karthik Subramanian
Peng Sun
Sayantan Sur
Sai Susarla
Le Sy Quoc
Alex Tabbal
Eddie Tan
Hwee-Xian Tan
Gabriel Tanase
Shao Tao
Mukarram Tariq
Joseph Teo
James Teresco
Matthew Thazhuthaveetil
Nathan Thomas
King Tin Lam
Srikanta Tirthapura
Andrei Tolstikov
Ken Tsang
Mark Tsang
David Tung Chong Wong
Abhishek Udupa
Alvin Valera
Ana Varbanescu
Srikumar Venugopal
Jakob Verbeek
Javier Verdu
Francesco Versaci
Stephane Vialle
Venkatram Vishwanath
Andy Wang
Chien-Min Wang
Jun Wang
Wei Wang
Tien-Hsiung Weng
Marek Wieczorek
Chih-Ying Wu
Jan-Jan Wu
Haiyong Xie
Ai Xin
Chao-Tung Yang
Laurence Yang
Yu Yang
Serhan Yarkan
Wai-Leong Yeow

Hao Yu
Wenjie Zeng
Mingze Zhang
Yan Zhang
Zizhan Zheng
Hu Zheng Qing
Jiazheng Zhou
Artur Ziviani

Table of Contents

Keynote Addresses

Session I: Performance Optimization

Session II: Parallel Algorithms and Applications

Session III: Scheduling and Resource Management

Session IV: Sensor Networks

Session V: Energy-Aware Computing

Session VI: Distributed Algorithms

Session VII: Communication Networks

Session VIII: Architecture

Extreme Computing on the Distributed European Infrastructure for Supercomputing Applications – DEISA

Wolfgang Gentzsch

Distributed European Initiative for Supercomputing Applications,
Duke University, USA

Abstract. Scientists' dream of accessing any supercomputer in the world, independently from time and space, is currently coming true, to perform even larger and more accurate computer simulations, at their finger tip. Today, high-speed networks transport data at the speed of light, middleware manages distributed computing resources in an intelligent manner, portal technology enable secure, seemless, and remote access to resources, applications, and data, and sophisticated numerical methods approximate the underlying mathematical equations in a highly accurate way. With the convergence of these core technologies into one complex service oriented architecture, we see the rise of large compute and data grids currently being built and deployed by e-Infrastructure initiatives such as DEISA, EGEE, NAREGI, and TERAGRID.

With the aid of one example, in this keynote presentation, we will elaborate on the Distributed European Infrastructure for Supercomputing Applications, DEISA, which recently entered its second phase. We will describe the system architecture, called the DEISA Common Production Environment (DCPE) and the DEISA Extreme Computing Initiative DECI attracting scientists all over Europe to use the networked supercomputing environment, and we will highlight a few impressive success stories from scientists who achieved breakthrough results so far which would not have been possible without such an infrastructure. Finally, we will summarize main lessons learned and provide some useful recommendations.

Biography: Wolfgang Gentzsch is Dissemination Advisor for the DEISA Distributed European Initiative for Supercomputing Applications. He is an adjunct professor of computer science at Duke University in Durham, and a visiting scientist at RENCI Renaissance Computing Institute at UNC Chapel Hill, both in North Carolina. From 2005 to 2007, he was the Chairman of the German D-Grid Initiative. Recently, he was Vice Chair of the e-Infrastructure Reflection Group e-IRG; Area Director of Major Grid Projects of the OGF Open Grid Forum Steering Group; and he is a member of the US President's Council of Advisors for Science and Technology (PCAST-NIT). Before, he was Managing Director of MCNC Grid and Data Center Services in North Carolina; Sun's Senior Director of Grid Computing in Menlo Park, CA; President, CEO, and CTO of start-up companies Genias and Gridware, and professor of mathematics and computer science at the University of Applied Sciences in Regensburg, Germany. Wolfgang Gentzsch studied mathematics and physics at the Technical Universities in Aachen and Darmstadt, Germany.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, p. 1, 2008.

Towards Networked Computers: What Can Be Learned from Distributed Computing?

David Peleg

Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Israel

Abstract. The talk will discuss some key ideas and concepts developed by the distributed computing community and examine their potential relevance to the development of networked computers.

Biography: David Peleg received the B.A. degree in 1980 from the Technion, Israel, and the Ph.D. degree in 1985 from the Weizmann Institute, Israel, in computer science. He then spent a post-doctoral period at IBM and at Stanford University. In 1988 he joined the Department of Computer Science and Applied Mathematics at The Weizmann Institute of Science, where he holds the Norman D. Cohen Professorial Chair of Computer Sciences. His research interests include distributed network algorithms, fault-tolerant computing, communication network theory, approximation algorithms and graph theory, and he is the author of a book titled “Distributed Computing: A Locality-Sensitive Approach,” as well as numerous papers in these areas.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, p. 2, 2008.

Computational Environments for Coupling Multiphase Flow, Transport, and Mechanics in Porous Media

Mary F. Wheeler

Center for Subsurface Modeling
Institute for Computational Engineering and Sciences
The University of Texas at Austin, USA

Abstract. Cost-effective management of remediation of contamination sites and carbon sequestration in deep saline aquifers is driving development of a new generation of subsurface simulators. The central challenge is to minimize costs of cleanup and/or maximize economic benefit from an environment whose properties are only poorly known and in which a variety of complex chemical and physical phenomena take place. In order to address this challenge a robust reservoir simulator comprised of coupled programs that together account for multicomponent, multiscale, multiphase flow and transport through porous media and through wells and that incorporate uncertainty and include robust solvers is required. The coupled programs must be able to treat different physical processes occurring simultaneously in different parts of the domain, and for computational accuracy and efficiency, should also accomodate multiple numerical schemes. In addition, this problem solving environment or framework must have parameter estimation and optimal control capabilities. We present a "wish list" for simulator capabilities as well as describe the methodology employed in the IPARS software being developed at The University of Texas at Austin. This work also involves a close cooperation on middleware for multiphysics couplings and interactive steering with Parashar at Rutgers University.

Biography: After 24 years at Rice University, Professor Mary Fanett Wheeler, a world-renowned expert in massive parallel-processing, arrived at The University of Texas in the Fall of 1995 with a team of 13 interdisciplinary researchers, including two associate professors, three research scientists, three postdoctoral researchers, and four Ph.D. students. Professor Wheeler is not completely new to UT, however, having received a B.S., B.A., and M.S. degrees from here before transferring to Rice for her Ph.D. under the direction of Henry Rachford and Jim Douglas, Jr. Drs. Rachford and Douglas, both of whom conducted some of the first applied mathematics work in modeling engineering problems, have greatly influenced her career.

With the oil industry's strong presence in Houston, she was at the right place at the right time to advance the leap from theoretical mathematics to practical engineering. She correctly theorized that parallel algorithms would spur a technological revolution, offering a multitude of applications in the fields of bioengineering, pharmaceuticals and population dynamics. Her reputation as a first class researcher has led to several national posts, including serving on the Board of Mathematical Sciences, on the Executive Committee for the NSF's Center for Research on Parallel Computation and in

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 3–4, 2008.

the National Academy of Engineering. Housed in the Texas Institute for Computational and Applied Mathematics (TICAM) on the UT campus, Professor Wheeler has brought a level of prominence to UT that many believe will bring us into the forefront of applied mathematics.

As Head of UT's new Center for Subsurface Modeling (CSM), which operates as a subsidiary of TICAM, Professor Wheeler and her team focus their computer-based research on finding solutions for societal and environmental dilemmas using computer simulations to help with, among other things, effective reservoir management within the oil and gas industry. Understanding contaminant movement and enhanced oil recovery techniques can save billions of dollars in cleanup as well as production over the next couple of decades. Hazardous waste cleanup is incredibly important to society, she believes, and is an area of study that has only begun to be explored.

Because of the complexity of the problems, Wheeler and her associates must obtain data about the geology, chemistry, and mechanics of a site before they can begin to construct algorithms to accurately depict a simulation. Hence, the interdisciplinary nature of the work, which no one individual within a single department could tackle on his/her own. Yet Professor Wheeler has indeed made great strides toward obtaining expertise in several disciplines key to the success of parallel computing. Indeed, she holds joint appointments in the Departments of Petroleum and Geosystems Engineering, Aerospace Engineering and Engineering Mechanics, and Mathematics. She is also the first woman to hold an endowed Chair in UT's College of Engineering (the Ernest and Virginia Cockrell Chair in Engineering).

Dr. Wheeler's own research interests include numerical solution of partial differential systems with application to the modeling of subsurface and surface flows and parallel computation. Her numerical work includes formulation, analysis and implementation of finite-difference/finite-element discretization schemes for nonlinear coupled pde's as well as domain decomposition iterative solution methods. Her applications include reservoir engineering and contaminant transport in groundwater and bays and estuaries. Current work has emphasized mixed finite-element methods for modeling reactive multi-phase flow and transport in a heterogeneous porous media, with the goal of simulating these systems on parallel computing platforms. Dr. Wheeler has published more than 100 technical papers and edited seven books. She is currently an editor of four technical journals and managing editor of Computational Geosciences. In 1998 she was elected to the National Academy of Engineering.

The Excitement in Parallel Computing

Laxmikant Kale

Department of Computer Science
University of Illinois at Urbana-Champaign, USA

Abstract. The almost simultaneous emergence of multicore chips and petascale computers presents multidimensional challenges and opportunities for parallel programming. Machines with hundreds of TeraFLOP/S exist now, with at least one having crossed the 1 PetaFLOP/s rubicon. Many machines have over 100,000 processors. The largest planned machine by NSF will be at University of Illinois at Urbana-Champaign by early 2011. At the same time, there are already hundreds of supercomputers with over 1,000 processors each. Adding breadth, multicore processors are starting to get into most desktop computers, and this trend is expected to continue. This era of parallel computing will have a significant impact on the society. Science and engineering will make breakthroughs based on computational modeling, while the broader desktop use has the potential to directly enhance individual productivity and quality of life for everyone. I will review the current state in parallel computing, and then discuss some of the challenges. In particular, I will focus on questions such as: What kind of programming models will prevail? What are some of the required and desired characteristics of such model/s? My answers are based, in part, on my experience with several applications ranging from quantum chemistry, biomolecular simulations, simulation of solid propellant rockets, and computational astronomy.

Biography: Professor Laxmikant (Sanjay) Kale has been working on various aspects of parallel computing, with a focus on enhancing performance and productivity via adaptive runtime systems, and with the belief that only interdisciplinary research involving multiple CSE and other applications can bring back well-honed abstractions into Computer Science that will have a long-term impact on the state-of-art. His collaborations include the widely used Gordon-Bell award winning (SC'2002) biomolecular simulation program NAMD, and other collaborations on computational cosmology, quantum chemistry, rocket simulation, space-time meshes, and other unstructured mesh applications. He takes pride in his group's success in distributing and supporting software embodying his research ideas, including Charm++, Adaptive MPI and the ParFUM framework. Prof. Kale received the B.Tech degree in Electronics Engineering from Benares Hindu University, Varanasi, India in 1977, and a M.E. degree in Computer Science from Indian Institute of Science in Bangalore, India, in 1979. He received a Ph.D. in computer science in from State University of New York, Stony Brook, in 1985.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, p. 5, 2008.

Improving Performance of Digest Caches in Network Processors

Girish Chandramohan[1,⋆] and Govindarajan Ramaswamy[2]

[1] Akamai Technologies India Pvt. Ltd.
[2] Supercomputer Education and Research Centre,
Indian Institute of Science, Bangalore 560 012, India
gchandra@akamai.com, govind@serc.iisc.ernet.in

Abstract. Digest caches have been proposed as an effective method to speed up packet classification in network processors. In this paper, we show that the presence of a large number of small flows and a few large flows in the Internet has an adverse impact on the performance of these digest caches. In the Internet, a few large flows transfer a majority of the packets whereas the contribution of several small flows to the total number of packets transferred is small. In such a scenario, the LRU cache replacement policy, which gives maximum priority to the most recently accessed digest, tends to evict digests belonging to the few large flows. We propose a new cache management algorithm called *Saturating Priority* (SP) which aims at improving the performance of digest caches in network processors by exploiting the disparity between the number of flows and the number of packets transferred. Our experimental results demonstrate that SP performs better than the widely used LRU cache replacement policy in size constrained caches. Further, we characterize the misses experienced by flow identifiers in digest caches.

1 Introduction

Network applications such as IP forwarding and packet classification involve complex lookup operations. These operations have to be performed at wire speeds and are the bottleneck in achieving faster processing rates in routers [1, 2]. Algorithmic techniques used for packet classification [2, 3] in routers need large tables and require multiple accesses to main memory. Architectural optimizations proposed for speeding up this processing involve caching the data structures used [1, 4, 5] or caching the results of the lookup [6, 7]. Cache based methods exploit temporal locality observed in Internet packets. Consequently, the efficiency of these schemes is dependent on the access characteristics observed in real traces from the Internet. A clear understanding of the cache access patterns is important in order to design a cache management policy for such applications.

Network processors (NPs) [8, 9, 10] have emerged as a viable option to implement network processing applications. The generic architecture of a network

[⋆] This study was conducted when the first author was at SERC, IISc.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 6–17, 2008.

processor consists of a number of simple in-order cores that perform the data plane operations [11]. These cores have support for multiple threads aimed at exploiting packet-level parallelism present in network applications. The cores have small data and instruction caches to speedup network processing applications. These caches are of size 2KB to 8KB typically [8, 9, 11].

In digest caching [7], a fixed, smaller length digest is obtained by hashing the fields used in packet classification. This digest is stored along with the flow class in a cache. Due to reduced the cache entry sizes, a digest cache of a few kilobytes is effective in maintaining a higher hit rate than a result cache.

Previous studies propose maintaining the cache entries with the LRU cache replacement strategy as it performs better than the LFU and probabilistic insertion policies [7]. LRU cache replacement works well in the presence of temporal locality. Internet flows however exhibit distinct characteristics which may be used to obtain higher performance from these constrained caches. In the Internet, a large number of flows have a single packet, whereas a small number of flows have multiple packets and contribute to the majority of the traffic [12]. From a caching perspective, we observe that flows with a single packet do not exhibit any (temporal) locality and hence it is not beneficial to store such entries in the digest cache. More importantly, these single packet flows may evict digests belonging to flows with multiple packets. We propose to exploit this characteristic of Internet flows to improve the performance of a digest cache by using a new cache replacement policy called *Saturating Priority* (SP). Under the SP cache replacement policy, a new digest entry is inserted in a cache set with the lowest priority. Its priority increases and reaches the maximum priority as more accesses are made to it. During cache replacement, the item with the lowest priority in the set is removed. Such a policy ensures that digests belonging to single packet flows do not replace multiple packet flow digests. We evaluate the miss rate with SP and LRU cache replacement policies using real traces collected from different sites in the Internet. The SP policy outperforms the LRU policy for all traces and cache sizes considered. For a 512-entry 4-way set associative cache, it covers 74% of the gap between LRU cache replacement and an oracle cache replacement policy which places digest entries in the cache only for flows that contain multiple packets.

Further, we characterize the misses incurred by a LRU managed digest cache and show that conflict misses are small compared to capacity and cold misses. This shows that although digest caches can reduce a majority of packet classification lookups, the size of the cache has to be substantially increased in order to reduce the misses to a small fraction.

The rest of the paper is organized as follows. In the next section, we present the necessary background for digest caches. In Sec. 3, we present the characteristics of Internet traffic in terms of flow size distribution and describe our Saturating Priority cache replacement algorithm. Sec. 4 deals with the performance results of the Saturating Priority algorithm. We present the related research proposals in this area in Sec. 5. Finally we conclude in Sec. 6.

2 Background

In this study, we consider a packet classification application [3] in which a tuple consisting of the source IP address and port, destination IP address and port and the protocol field are used to identify packets belonging to a particular network flow. This 5 tuple, called the flow identifier, is used to map the packets of a flow to a particular traffic QoS class. This methodology may also be used in packet forwarding application where the port on which the packet has to be forwarded is stored along with the flow identifier. In network address translation (NAT) application, the digest cache can be used to lookup the address translation information. A digest cache is a generic method for improving the table lookup rate, a commonly used operation in network processing applications.

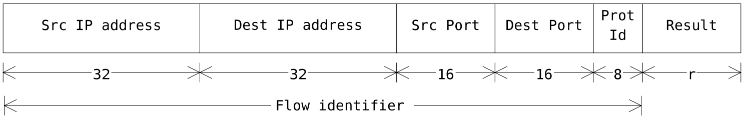


Fig. 1. Result cache entry for packet classification

In case of packet classification for IPv4, result caching involves storing the 104 bit 5-tuple along with the QoS or forwarding information of size r bits as shown in Fig. 1. The memory size required to realize such a cache with sufficiently high hit rate could be large due to the large size of the entries. A recent proposal has been to use a smaller digest of the 5-tuple instead of the actual values in the fields [7].

2.1 Operation of a Digest Cache

We now describe the operation of the digest cache which was presented by Chang et al. [7]. In case of packet classification, a digest cache works by using a hashing algorithm on the 104-bit 5 tuple to generate a 32-bit hash [1]. The least significant s bits are used to index into a set associative cache of 2^s sets. The remaining $(32 - s)$ bits of the hash are used for tag check, after which a cache hit or miss is known. Each cache block stores the r-bit result which is either the classification information or the forwarding information, depending on the application, for which the digest cache is used. Each cache block stores the result of only one tuple as spatial locality is low in these applications. In case of a miss, the processing proceeds by using a full classification algorithm. The digest that missed in the cache replaces another entry in the set using a cache eviction policy that chooses the victim. Steps involved in accessing a digest cache are shown in Fig. 2.

The digest cache acts as a filter, servicing the frequently accessed digests. Only those packets whose digests miss, go through the slower packet classification

[1] NPs such as the IXP2400 and IBM PowerNP have a hash unit which may be used to compute the digest.

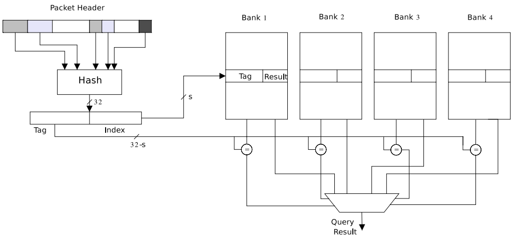


Fig. 2. Accessing an entry in a 4-way set associative digest cache

algorithm. A higher hit rate in the digest cache is critical for higher classification rates as it decreases the number of packets going through the slower packet classification step [5].

Each entry in the digest cache consists of a tag of $(32 - s)$ bits and r bits of classification information. Thus the total size of a digest cache with 2^s sets and k blocks per set (associativity) is

$$Digest\ Cache\ Size = \frac{((32 - s + r) * 2^s * k)}{8}\ bytes \qquad (1)$$

The size of a cache with different number of entries and associativities are shown in Table 1. Here, we assumed that 1 byte IP lookup or classification information is stored along with the digest. This is sufficient for various diff serve classes for which classification has to be performed.

Table 1. Cache sizes for different number of entries and associativities

Entries	4-way assoc.	8-way assoc.
512	2112 bytes	2176 bytes
1024	4096 bytes	4224 bytes
2048	7936 bytes	8192 bytes

The IXP 2400 has 2560 bytes of local memory, a 16 entry content accessible memory and a hash unit that may be used to implement a digest cache under software control [8]. The CAM supports LRU replacement. However, we show that using a different cache replacement policy could improve the performance of the cache. The local cache in each processing core of an NP is small, around 2KB. In this study, we consider the performance of digest caches in a network processor environment and consider cache sizes up to 8KB.

3 Improving the Performance of Digest Caches

Li et al., [6] evaluate digest caches with different associativities, hash functions and cache management policies. They propose a probabilistic insertion policy which only partially addresses the problem of the presence of large number of flows. The benefit of this algorithm is comparable to the LRU replacement policy. In fact, the probabilistic replacement policy and LFU replacement policies have a higher miss rate than the LRU cache replacement with two of the edge traces that they considered. On the other hand, we first study the distribution of flow lengths in the Internet and their effect on LRU cache replacement. The number of flows in the Internet, their lengths and rates influences the performance of the digest cache. We then propose SP cache replacement policy which exploits the widely observed disparity in flow sizes to find cache entries for replacement.

3.1 Flow Length Distribution in the Internet

An Internet packet flow is considered active from the time it sends the first packet until it is idle, i.e, it does not send a packet, for 1 second. A similar definition is used in [12][2]. Flow length is defined as the number of packets sent when a flow is active. The traces from [13] that we use in our study are shown in Table 2. Only Abilene trace has packets that are in both directions[3]. However we show that irrespective of the direction of the flow of packets, there is a disparity in the flow lengths and the number of packets transferred by these flows.

Table 2. Traces from the Internet used in this study

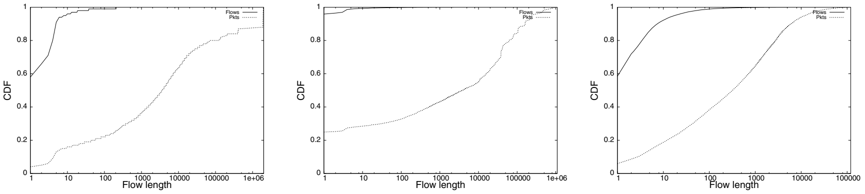
Trace (Label)	Type	Trace Direction	Files
Abilene Indianapolis Abilene router (Abilene)	Core	Bidirectional	I2K-1091235140-1.erf.gz to I2K-1091308171-2.erf.gz
National Center for Atmospheric Research (NCAR)	Edge	Unidirectional	20031229-223500.gz to 20031229-221000.gz
Front Range Gigapop (FRG)	Edge	Unidirectional	FRG-1133754905-1.tsh.gz to FRG-1133818534-1.tsh.gz
Pittsburgh Supercomputing Center (PSC)	Edge	Unidirectional	PSC-1145580225-1.tsh.gz to PSC-1145742328-1.tsh.gz
San Diego Supercomputer Center to Abilene connection (SDA)	Edge	Unidirectional	SDA-1141865872.erf.gz to SDA-1142027975.erf.gz

For the three traces, Abilene, NCAR and SDA, Fig. 3 shows the cumulative percentage of flows having packets less than the value on the x-axis. The graph also shows the cumulative percentage of the packets that are transferred by these flows. These traces have 3.1 million, 15.4 million and 4.4 million flows respectively. In NCAR trace, more than 95% of the flows have only a single packet. Abilene and SDA traces have 58% single packet flows. Fig. 3 also shows

[2] In [12], the authors use a time out of 60 seconds. Even with a time out of 1 second, there are about 6000 concurrent flows in all our traces. This is more than the number of entries in the digest cache.

[3] The packets were merged according to their timestamp to obtain a proper interleaving of the packets as seen by the router.

the cumulative percentage of the traffic (in terms of packets) contributed by flows of different sizes. It can be seen that, irrespective of the direction of the traces, single packet flows, which are a significant part of the total number of flows, contribute less than 6% of the packets transferred in Abilene and SDA traces. In NCAR trace, single packet flows transfer 25% of the total packets. Less than 0.2% of the flows have more than 1000 packets, but they transfer 52% of the packets on an average. Thus Internet traffic exhibits the phenomenon of mass-count disparity [14], i.e. a few *important* flows transfer a *significant* number of packets. From the above observations, we see that a large number of flows in the Internet have only a single packet. These flows get entries into the digest cache, but will never be accessed again. Presence of a large number of such flows has a detrimental effect on the performance of the digest cache as they tend to evict the digests of few flows that contain a large number of packets.



(a) Abilene trace (b) NCAR trace (c) SDA trace

Fig. 3. Disparity in the flow lengths and packets transferred

3.2 A New Replacement Algorithm for Digest Caches

The above observations provide an insight into the functioning of the digest cache in a network processor. The cache has to service a large number of flows, but only a small fraction of the flows have multiple packets that are accessed again. From Fig. 3, we infer that the cache replacement policy of a digest cache must strive to retain the digests belonging to large flows within the cache while preventing the single packet flows from occupying entries in the cache.

Given the large number of single packet flows in the Internet and the small cache size, an LRU managed digest cache may suffer from degraded performance due to cache pollution. Previous studies using digest caches used LRU cache replacement strategy as it performed better than LFU and probabilistic insertion policies [6, 7]. LRU cache management gives preference to the most recently accessed entry in the set and puts it in the most-recently-used (MRU) location. In case of network processing applications, the most recently accessed digest usually belongs to a single packet flow and it will never be accessed again. But such digests stay in the cache until they are slowly pushed out by the LRU replacement algorithm. This suggests that LRU may not be the best cache management policy.

In order to overcome this effect, we propose a *Saturating Priority* cache replacement policy that exploits the disparity between the number of flows and the residency of the digests in the cache. Each entry in a set has a priority that increases each time the entry is accessed, until it reaches a maximum value. Further accesses to the same entry do not increase its priority. Whenever an entry with a lower priority is accessed, it swaps its priority with an entry that has the next higher precedence. A new entry is added to a set with the least priority, after evicting the item with the lowest priority.

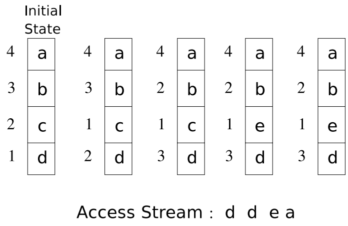


Fig. 4. Saturating Priority cache replacement policy

Fig. 4 illustrates the change in priorities of the entries in a 4-way set associative set. The set initially contains the digests marked *a,b,c* and *d*. Their priorities are also marked in the figure. Here, a larger number signifies a higher priority. When digest d is accessed two times, it swaps its priority with digests c and b respectively. It thus has a priority of 3. The priority of digests b and c decrease. Digest c now has the lowest priority in the set. As a result, the miss caused by digest e evicts digest c from the cache. The last access in the sequence, access to digest a, does not increase its priority as it has already got the maximum priority.

SP cache replacement evicts any entry that was brought into the cache, but is not subsequently accessed. Entries that are accessed only a few times are likely to be evicted. Also in a digest cache, entries are accessed a number of times when the a flow is active, but the accesses stop when the flows end. The cache replacement policy proactively removes such entries as their priority decreases rapidly.

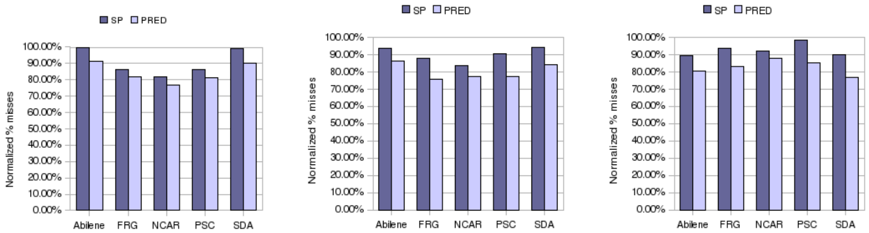
SP scheme can be implemented in a d way associative cache by maintaining two $\log_2(d)$ bit pointers per cache entry, which point to cache entries with immediately higher and lower priority. When the priority of an element changes, the pointers in the (at most four) affected cache entries can be updated in parallel.

4 Performance Evaluation

As mentioned in Sec. 1, IP lookup or packet classification is the bottleneck in achieving higher processing rates in routers [1, 2]. By reducing the miss rate of digest caches, higher processing rates can be achieved as the packets are processed from the cache entries instead of the slower off-chip memory. We therefore compare the miss rate of SP scheme with that of the widely used LRU scheme.

In order to understand the maximum improvement in performance of the digest cache that is possible by preventing the detrimental effect of single packet flows, we implemented an oracle cache management policy that inserts a digest into the cache only when it has more than one access in a window of 10,000 accesses in the future. The cache entries are managed with a LRU policy. We call this PRED policy as a simple predictor may be able to predict single packet flows reasonably well.

We consider cache sizes ranging from 2KB to 8KB because as explained in Sec. 1 these are the typical memory sizes available in network processors to implement digest caches. 32-bits from the MD5 hash of the flow identifier is used to obtain the digest. The cache sizes for different configurations are shown in Table 1.



(a) 512 entries, 4-way set (b) 1024 entries, 4-way set (c) 2048 entries, 4-way set

Fig. 5. Normalized misses for a 4-way associative digest cache

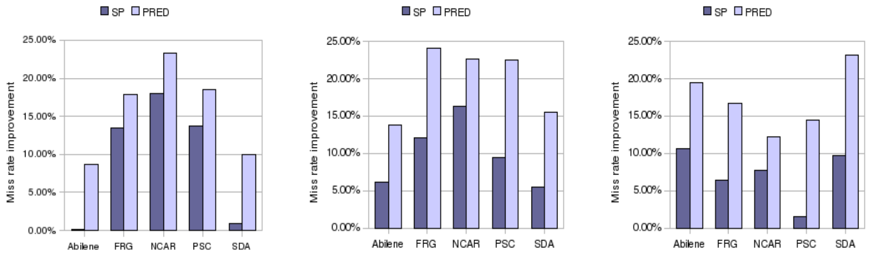
For the traces listed in Table 2. Fig. 5 shows the percentage of misses with SP and PRED replacement policies. The misses are normalized wrt. misses incurred with LRU replacement for 4-way set associative caches. The SP policy performs better than the LRU cache replacement policy in terms of miss rates for all 4-way set associative caches. As expected, the PRED cache management policy has lower miss rate than the other two policies.

Table 3 shows the miss rates for LRU and SP policies for 4-way and 8-way associative caches. We see that for almost all the configurations, SP replacement for a 4-way set associative cache has a lower miss rate than LRU replacement for

Table 3. Cache miss rates with SP and LRU policies

Trace	512 Entries				1024 Entries				2048 Entries			
	4 way		8 way		4 way		8 way		4 way		8 way	
	LRU	SP	LRU	SP	LRU	SP	LRU	SP	LRU	SP	LRU	SP
Abil.	32.6%	32.5%	32.2%	32.8%	29.3%	27.5%	29.1%	26.7%	24.9%	22.2%	24.7%	21.3%
FRG	18.6%	16.1%	18.2%	15.3%	11.9%	10.5%	11.0%	9.8%	7.4%	7.0%	6.7%	6.7%
NCA.	42.1%	34.5%	42.1%	32.8%	35.4%	29.6%	34.6%	28.4%	29.7%	27.3%	28.6%	27.2%
PSC	20.8%	17.9%	19.9%	16.9%	13.0%	11.8%	11.8%	11.1%	8.2%	8.1%	7.4%	8.1%
SDA	49.8%	49.4%	49.1%	50.0%	40.2%	37.9%	39.2%	37.6%	29.7%	26.8%	28.9%	25.8%

a 8-way set associative cache. In SP cache replacement, the maximum priority that an element can get is equal to its set associativity. An element that attains higher priority occupies the cache for a longer duration before it is evicted from the set. As a result, when a flow becomes inactive, it takes longer to be evicted from the cache. This explains the low reduction in miss rate with SP policy for caches with higher associativity.



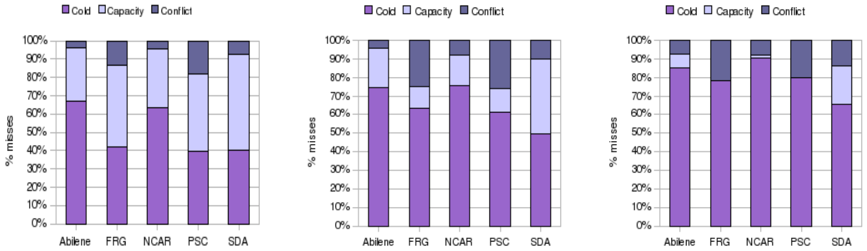
(a) 512 entries, 4-way set (b) 1024 entries, 4-way set (c) 2048 entries, 4-way set

Fig. 6. Miss rate improvement over LRU replacement for 4-way assoc. caches with SP and PRED policies

Fig. 6 shows the improvement in miss rate over LRU scheme for 4-way set associative caches with SP and PRED cache replacement. The NCAR trace shows 18% improvement in miss rate for a 512-entry cache and 8% improvement for a 2048-entry cache. For a 512-entry cache, with the FRG, NCAR and PSC traces, SP covers more than 74% of the gap between the LRU and PRED cache management policy. Even for higher cache sizes, SP covers substantial gap between LRU and PRED replacement. With the PSC trace, SP cache replacement does not show much improvement with large caches, however the miss rate for this trace is already low. For a 512-entry cache with the SDA trace, SP replacement policy has a small improvement over LRU replacement policy. But it has 10% improvement over LRU replacement policy for a 2048-entry cache, covering 41% of the gap between LRU and PRED policies. For 512-entry and 1024-entry caches, SP shows more than 10% miss rate improvement of an average whereas for a 2048-entry cache, the average improvement is 7.08%.

With a 2048-entry cache, the PSC trace with SP cache replacement shows a slightly higher miss rate of 8.1% compared to LRU cache replacement with a miss rate of 7.4%. We observe that with this cache size, the PSC trace has insignificant capacity misses (refer Fig. 7). In this case, the SP cache replacement policy that evicts the most recently accessed digests incurs slightly higher misses as the digests are removed before their priority can increase. But with smaller caches such as those seen in NPs, SP policy has lower miss rate than LRU cache replacement.

We see that the Abilene, NCAR and SDA traces suffer more than 13% miss rate even for a 2048-entry cache with LRU cache replacement. In order to understand the reason for this, we classified the misses in a LRU managed cache as cold, capacity and conflict misses. Misses due to accesses that were not present in a window of previous 10,000 accesses are classified as cold misses. We used this definition of cold misses because when network flows stop, their digests are evicted from the cache. When a packet with the same digest is seen next, it is considered a new flow. Accesses to digests that are present in the window of previous 10,000 accesses but not in a fully associative cache are classified as capacity misses. Conflict misses are those which occur when a set associative cache is used instead of a fully associative cache of the same size.



(a) 512 entries, 4-way set (b) 1024 entries, 4-way set (c) 2048 entries, 4-way set

Fig. 7. Types of misses in digest caches

Conflict misses may be reduced by using a better cache placement and management policy. From Fig. 7 we observe that for a 1024-entry, 4-way set associative digest cache less than 10% of the misses are due to cache conflicts in case of Abilene, NCAR and SDA traces. Traces that already have a high hit rate, such as FRG and PSC, have about 25% conflict misses. This observation shows that the cold and capacity misses dominate the misses in a digest cache. As expected, for larger caches, the ratio of capacity misses decreases but the number of cold misses does not decrease. This is mainly because of the large number of small flows (refer Fig. 3). As a result, continuously increasing the cache size leads to small improvements in performance. Instead, it may be worthwhile to use better algorithmic or data structure caching approaches to improve the hit rate.

5 Related Work

Zhang et al. [12] use traces from different locations in the core and edge of the Internet to study the characteristics of Internet traffic flows. Disparity in the flow lengths is shown to be more drastic than the disparity in the rate of the flows. They also show that there is a correlation between the size of the flow and its rate. This disparity in the number of packets present in a few flows inspired us to propose a new cache management scheme for digest caches.

Feitelson [14] proposed metrics to quantify mass-count disparity, a phenomenon seen in a number of computer applications such as network traffic, job time distribution in operating systems, file size distribution. In [15], Feitelson et al. use the same phenomenon to design a filter for identifying and retaining common addresses in a direct mapped cache L1 cache of a superscalar processor. Since a few addresses are accessed a large number of times, the authors show that a randomly chosen reference to the cache belongs to an address that is accessed many times.

Mudigonda et al. [4] propose the use of caches for data structures used in network processing applications as it benefits a large number of such programs. Similarly, Gopalan et al. [5], propose an intelligent mapping scheme to reduce conflict misses in IP lookup data structure caches and to enable frequent updates of the routing table. On the other hand, we exploit the traffic patterns observed in real traces to improve the effectiveness of small digest caches for packet classification application. The insight gained from the traffic patterns can also be applied to data structure caches. We leave this to future work.

Qureshi et al. [16] propose a LRU insertion policy (LIP) for L2 caches in general purpose processors which, like SP, inserts the cache lines in the least recently used location in the set instead of the MRU location. Lines are moved to the MRU location in case they are accessed in the LRU location whereas the SP policy allows the cache entries to slowly percolate to the MRU position. LIP is aimed at applications that have a larger working set than the cache size. For applications that have a cyclic reference pattern, it prevents thrashing of the cache by retaining some entries in the cache so that they contribute to cache hits. On the other hand, we observe that the large disparity in the flow sizes in the internet leads to poor performance of LRU managed result caches in network applications. In SP, the priorities are managed such that cache replacement policy can recognize digests belonging to large flows.

6 Conclusions

Digest caches provide an effective mechanism to reduce the number of expensive off-chip lookups. However, they suffer from poor performance due to the large number of single packet flows in the Internet. We proposed a new cache replacement policy, called Saturating Priority, that overcomes the detrimental effects of these flows on the performance of digest caches. This policy performs better the widely used the LRU cache replacement policy for space constrained caches. We showed that Saturating Priority covers nearly three fourth of the gap between the LRU cache replacement and the oracle cache replacement policy, which places an entry in the cache only when there are multiple packets in the flow. Further, we showed that the majority of the misses in a digest are cold misses. This emphasizes the need for algorithmic innovations to improve packet classification performance.

References

1. Baer, J.L., Low, D., Crowley, P., Sidhwaney, N.: Memory Hierarchy Design for a Multiprocessor Look-up Engine. In: PACT 2003: Proc. of the 12th Intl. Conf. on Parallel Arch and Compilation Techniques, p. 206. IEEE Computer Society, Los Alamitos (2003)

2. Taylor, D.E.: Survey and taxonomy of packet classification techniques. ACM Comput. Surv. 37(3), 238–275 (2005)
3. Gupta, P., McKeown, N.: Algorithms for packet classification. IEEE Network 12(2), 24–32 (2001)
4. Mudigonda, J., Vin, H.M., Yavatkar, R.: Overcoming the memory wall in packet processing: hammers or ladders? In: ANCS 2005: Proc. of the 2005 symp. on Architectures for networking and comm. sys., pp. 1–10. ACM Press, New York (2005)
5. Gopalan, K., cker Chiueh, T.: Improving route lookup performance using network processor cache. In: Supercomputing 2002: Proc. of the 2002 ACM/IEEE conf. on Supercomput., pp. 1–10. IEEE Computer Society Press, Los Alamitos (2002)
6. Li, K., Chang, F., Berger, D., chang Feng, W.: Architectures for packet classification caching. In: ICON 2003: The 11th IEEE Intl. Conf. on Networks, September 28-October 1, 2003, pp. 111–117 (2003)
7. Chang, F., Chang Feng, W., Chi Feng, W., Li, K.: Efficient packet classification with digest caches. In: NP 3: Workshop on Network Procs. and Appns. (2004)
8. Intel: Intel IXP2400 Network Processor Hardware Reference Manual (November 2003)
9. Allen, J.R., et al.: IBM PowerNP network processor: Hardware, software, and applications. IBM J. Res. Dev. 47(2-3), 177–193 (2003)
10. C-Port Corporation: C-5 Network Processor D0 Architecture Guide (2001)
11. Wolf, T., Franklin, M.A.: Design tradeoffs for embedded network processors. In: Schmeck, H., Ungerer, T., Wolf, L. (eds.) ARCS 2002. LNCS, vol. 2299, pp. 149–164. Springer, Heidelberg (2002)
12. Zhang, Y., Breslau, L., Paxson, V., Shenker, S.: On the characteristics and origins of internet flow rates. In: SIGCOMM 2002: Proc. of the 2002 conf. on Applications, technologies, arch. and protocols for comp. commn., pp. 309–322. ACM, New York (2002)
13. National Laboratory for Applied Network Research, `http://pma.nlanr.net`
14. Feitelson, D.G.: Metrics for mass-count disparity. In: MASCOTS 2006: Proc. of the 14th IEEE Intl. Symp. on Modeling, Analysis and Simulation, Washington, DC, USA, pp. 61–68. IEEE Computer Society, Los Alamitos (2006)
15. Etsion, Y., Feitelson, D.G.: L1 cache filtering through random selection of memory references. In: PACT 2007: Proc. of the 16th Intl. Conf. on Parallel Arch and Compilation Techniques, pp. 235–244. IEEE Computer Society, Los Alamitos (2007)
16. Qureshi, M.K., Jaleel, A., Patt, Y.N., Steely, S.C., Emer, J.: Adaptive insertion policies for high performance caching. In: ISCA 2007: Proc. of the 34th annual Intl. symp. on Comp. arch., pp. 381–391. ACM, New York (2007)

Optimization of BLAS on the Cell Processor

Vaibhav Saxena[1], Prashant Agrawal[1], Yogish Sabharwal[1], Vijay K. Garg[1], Vimitha A. Kuruvilla[2], and John A. Gunnels[3]

[1] IBM India Research Lab, New Delhi
{vaibhavsaxena,prashant_agrawal,ysabharwal,vijgarg1}@in.ibm.com
[2] IBM India STG Engineering Labs, Bangalore
vimitha.k@in.ibm.com
[3] IBM T. J. Watson Research Center, Yorktown Heights, NY
gunnels@us.ibm.com

Abstract. The unique architecture of the heterogeneous multi-core Cell processor offers great potential for high performance computing. It offers features such as high memory bandwidth using DMA, user managed local stores and SIMD architecture. In this paper, we present strategies for leveraging these features to develop a high performance BLAS library. We propose techniques to partition and distribute data across SPEs for handling DMA efficiently. We show that suitable preprocessing of data leads to significant performance improvements when the data is unaligned. In addition, we use a combination of two kernels – a specialized high performance kernel for the more frequently occurring cases and a generic kernel for handling boundary cases – to obtain better performance. Using these techniques for double precision, we obtain up to 70–80% of peak performance for different memory bandwidth bound level 1 and 2 routines and up to 80–90% for computation bound level 3 routines.

Keywords: Cell processor, multi-core, Direct Memory Access (DMA), BLAS, linear algebra.

1 Introduction

The Cell Broadband Engine, also referred to as the Cell processor, is a multi-core processor jointly developed by Sony, Toshiba and IBM. The Cell is a radical departure from conventional multi-core architectures – combining a conventional high-power PowerPC core (PPE) with eight simple Single-Instruction, Multiple-Data (SIMD) cores, called Synergistic Processing Element (SPE) in a heterogeneous multi-core offering. It offers extremely high compute-power on a single chip combined with a power-efficient software-controlled memory hierarchy. The theoretical peak performance of each SPE for single precision floating point operations is 25.6 GFLOPS leading to an aggregate performance of 204.8 GFLOPS for 8 SPEs. The theoretical peak performance for double precision is 12.8 GFLOPS per SPE and 102.4 GFLOPS aggregate. Each SPE has 256 KB of Local Store for code and data. An SPE cannot directly access

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 18–29, 2008.

the data stored in an off-chip main memory and explicitly issues Direct Memory Access (DMA) requests to transfer the data between the main memory and its local store. Access to the external memory is handled via a 25.6 GB/s Rambus extreme data rate (XDR) memory controller. The PPE, eight SPEs, memory controller and input/output controllers are connected via the high bandwidth Element Interconnect Bus (EIB) [9].

Distinctive features of the Cell such as the XDR memory subsystem, coherent EIB interconnect, SPEs, etc. make it suitable for computation and data intensive applications. There has been a considerable amount of work that has demonstrated the computational power of the Cell processor for a variety of applications, such as dense matrix multiply [16], sparse matrix-vector multiply [16], fast Fourier transforms [3], sorting [6], ray tracing [4] and many others.

Basic Linear Algebra Subprograms (BLAS) is a widely accepted standard for linear algebra interface specifications in high-performance computing and scientific domains, and forms the basis for high quality linear algebra packages such as LAPACK [1] and LINPACK [5]. BLAS routines [13] are categorized into three classes – level 1 routines (vector and scalar operations), level 2 routines (vector-matrix operations) and level 3 routines (matrix-matrix operations).

BLAS has been tuned and optimized for many platforms to deliver good performance, e.g. ESSL on IBM pSeries and Blue Gene [11], MKL for Intel [12], GotoBLAS on a variety of platforms [7], etc. Successful efforts have also been made towards automatic tuning of linear algebra software (ATLAS) [2] to provide portable performance across different platforms using empirical techniques. Some of these portable libraries give good performance when executed on the Cell PPE. However, given the unique architecture of the Cell processor and the SPE feature set, specialized code needs to be designed and developed for obtaining high performance BLAS for the Cell processor. Williams et al. [16], [15] have discussed optimization strategies for the general matrix-multiply routine on the Cell processor, obtaining near-peak performance. However, existing literature and optimization strategies of linear algebra routines on the Cell make simplified assumptions regarding the alignment, size, etc. of the input data. A BLAS library needs to address many issues for completeness, such as different alignments of the input vectors/ matrices, unsuitable vector/ matrix dimension sizes, vector strides, etc., that can have significant impact on the performance.

In this paper, we discuss the challenges and opportunities involved in optimizing BLAS for the Cell processor. We focus on the optimization strategies used for producing the high performance BLAS library that is shipped with the Cell Software Development Kit (SDK). The library consists of single and double precision routines ported to the PPE; a selected subset of these routines have been optimized using the SPEs. The routines conform to the standard BLAS interface at the PPE level. This effort, of offering a high performance BLAS library, is the first of its kind for the Cell. We propose techniques to partition and distribute data across SPEs for handling DMA efficiently. We show that suitable pre-processing of data leads to significant performance improvements when the data is unaligned. In addition, we use a combination of two kernels – a specialized high performance kernel for the more frequently occurring cases and a generic kernel for handling boundary cases – to obtain better performance.

The rest of the paper is organized is as follows. In Section 2 we discuss the challenges and opportunities that the Cell offers with respect to BLAS. In Section 3, we discuss the optimization strategies followed by performance results in Section 4. We conclude in Section 5 with a brief discussion of the ongoing and future planned work.

2 Challenges and Opportunities

On the Cell processor, the memory hierarchy of the PPE is similar to conventional processors whereas SPEs have a distinctive three-level hierarchy: (a) 128×128-bit unified SIMD register file, (b) 256 KB of local store memory, and (c) shared off-chip main memory. Each SPE works only on the code and data stored in its local store memory and uses DMA transfers to move data between its local store and the main memory (or the local stores of other SPEs).

These DMA transfers are asynchronous and enable the SPEs to overlap computation with data transfers. Although the theoretical peak memory bandwidth is 25.6 GB/s, the effective bandwidth obtained may be considerably lower if the DMA transfers are not setup properly. This can degrade performance of BLAS routines, particularly level 1 and level 2 routines, which are typically memory bandwidth bound.

DMA performance is best when both source and destination buffers are 128-byte (one cache line) aligned and the size of the transfer is a multiple of 128 bytes. This involves transfer of full cache lines between main memory and local store. If the source and destination are not 128-byte aligned, then DMA performance is best when both have the same quadword offset within a cache line. This affects the data partitioning strategy. Typically, an SPE works on blocks of the input data by iteratively fetching them from main memory to its local store, performing required operation on these blocks and finally storing back the computed data blocks to main memory. Therefore, it is important to partition the input data in a manner such that the blocks are properly aligned so that their DMA transfers are efficient.

Transfer of unaligned or scattered data (e.g. vectors with stride greater than 1) may result in the use of DMA lists. However, direct (contiguous) DMA transfers generally lead to better bandwidth utilization in comparison to DMA list accesses as every list element consumes at least 128 bytes worth of bandwidth, independent of the size of the transfer. To illustrate this, consider a block of size $16x + 12$ bytes starting at a 128 byte aligned address. DMA transfers can be done in units of 1, 2, 4, 8 and multiple of 16 bytes starting at memory addresses that are 1, 2, 4, 8 and 16 byte aligned, respectively. One way of transferring this block is to construct a DMA list that has 3 list elements, one each for (1) the $16x$ byte aligned part, (2) the 8 byte part and (3) the 4 byte part. As each transfer consumes 128 bytes worth of bandwidth, there may be close to 128 bytes worth of bandwidth loss for each transfer. A better strategy is to transfer some extra bytes at the tail, making the transfer size a multiple of 16 bytes so that a direct DMA transfer can be used. When fetching data, the extra fetched bytes can be discarded by the SPE. However this strategy cannot be used for writing data back to main memory as it can lead to memory inconsistencies. Hence, DMA lists need to be used for writing back unaligned data.

Selection of an appropriate block size is also critical for high performance. Large data block size not only improves DMA efficiency due to larger data transfers but also results in sufficient computations to hide overlapped DMA latencies. However, the data block size is limited by 256 KB of SPE local store.

3 Optimization Strategies

Algorithmic and architecture specific optimizations of linear algebra libraries such as BLAS and LAPACK, have been well studied [1,2,7,12,13]. However, several factors have to be taken into consideration when applying these proven strategies on the Cell. Besides, new techniques are required for enabling high performance of routines like BLAS on the Cell, as discussed in Section 2. In this section we discuss the different strategies used for optimizing BLAS on the Cell. It should be noted that these strategies are targeted for a single Cell processor, large data sets, column-major matrices and huge memory pages (16 MB).

3.1 Data Partitioning and Distribution

Data partitioning and distribution are a critical part of designing linear algebra subprograms on multi-cores. The proposed strategy for data partitioning and distribution differs across the three categories of the BLAS routines. For the memory bandwidth bound level l and level 2 routines, data partitioning is carried out with an objective to get close to the peak memory bandwidth, whereas for the computation bound level 3 routines the objective is to get close to the peak computation rate.

BLAS Level 1 Routines: Level 1 routines typically operate on one or two vectors and produce as output a vector or a scalar. The goal is to partition the data into equal-sized blocks that can be distributed to the SPEs with each SPE getting roughly an equal number of blocks. In our strategy, when the output is a vector, the output or the I/O vector (data that is both read and updated) is divided by taking into considerations the memory alignment of the blocks such that they are 128-byte aligned, are multiple of 128 bytes and large enough (16 KB – the maximum transfer size for a single DMA operation). This ensures that the DMA writes from local store to main memory can be performed without the need for DMA lists. These blocks are then divided (almost) equally, among the SPEs, with each SPE getting a contiguous set of blocks. The other input vector (if any) is divided with respect to the output or I/O vector, without considering the memory alignment, such that their blocks have the same range of elements. In cases where the output is a scalar (e.g. in DDOT), partitioning with memory alignment considerations can be carried out for any of the vectors.

When the vector being partitioned does not start or end on a 128-byte boundary, there may be small parts of the vector at the start (head) and the end (tail) that do not satisfy the alignment and size criteria mentioned above. These are handled directly on the PPE.

In the case where access for one or more vectors is strided, the size of each block is restricted to 2048 elements (the maximum number of DMA transfers that can be specified in a single DMA list operation).

BLAS Level 2 Routines: Level 2 routines perform matrix-vector operations and their output can either be a vector or a matrix. The complexity of these routines is determined by the memory bandwidth requirements for fetching/storing the matrix. Thus, data partitioning and distribution for these routines is done keeping in mind efficient DMA considerations for the matrix. The column-major matrix is divided into rectangular blocks which are distributed among the SPEs. The SPEs typically operate on one block in an iteration. A block is fetched using a DMA list where each list element transfers one column of the block. To improve the efficiency of the DMA, column sizes of the block should be large and multiples of 128 bytes. The block dimensions are appropriately chosen depending on the number of vectors used and SPE local store size.

If the output is a vector and there are two vectors – an input and a I/O vector (e.g. in DGEMV), the I/O vector is divided into blocks by taking memory alignment into consideration and distributed to the SPEs, as it is done for level 1 routines. Each SPE fetches an I/O vector block, iteratively fetches the blocks of the matrix and the input vector required for the computation, carries out the computation and writes back the I/O vector block to the main memory. If there is only one I/O vector (e.g. in DTRMV), a block of elements cannot be updated until all the computations involving it are completed. To resolve this dependency, a copy of the vector is created and is used as the input vector. The SPEs can then independently update the blocks of the output vector.

BLAS Level 3 Routines: Level 3 routines perform matrix-matrix operations and are computationally intensive. Thus, the key consideration in data partitioning and distribution for these routines is computational efficiency. The matrices are partitioned into square blocks (to maximize computations in order to hide DMA latencies) instead of rectangular blocks (which are more DMA efficient) as in the case of level 2 routines. The blocking factor of the matrices is decided based on factors such as SPE local store size and the number of input and output matrices being operated upon. Another important factor influencing the blocking factor is that when up to 16 SPEs are used on multi-Cell processor platforms, such as the IBM BladeCenter, the block size should result in sufficient computations so that the routine does not become memory bandwidth bound. Taking all these constraints into consideration, we have determined that a blocking factor of 64×64 can be used with the given memory constraints and is sufficient to keep the level 3 routines computation bound, even with 16 SPEs.

When there are no dependencies in the computation of the output matrix blocks (e.g. in DGEMM), these blocks are distributed across the SPEs and each SPE determines at runtime the output matrix block to process. This dynamic distribution of the blocks ensures a better load balancing across the SPEs. An SPE fetches an output matrix block, iteratively fetches the input matrices blocks required for the computation of the output block, carries out the computation and stores back the computed block to main memory. Since input matrix blocks are used multiple times in the computation of different output matrix blocks, the input matrices are reformatted before the computation (see section 3.2 for more details) to improve the DMA efficiency for the transfer of these blocks.

In the case where there are dependencies in the computation of the output matrix blocks, sets of the blocks are distributed across the SPEs such that the computation across these sets are independent as much as possible. The order of computation of the blocks within a set is routine specific, e.g. in the case of TRSM while computing $B \leftarrow A^{-1} \cdot B$, where A is a lower triangular matrix, dependencies exists in the computation of the elements along a column but there is no dependency among elements in different columns. Therefore the columnsets[1] can be computed independently. Thus for TRSM, the columnsets of the output matrix are distributed across the SPEs and the blocks are processed in the top-down order within a columnset. Similar distribution can be used for other input parameter combinations as well. The SPEs determine at runtime the sets they should process.

For particular combinations of input parameters, we carry out the complete computation on the PPE if it is more beneficial – for instance when the matrix/vector dimensions are so small that SPE launching overheads exceed computation time.

3.2 Efficient DMA Handling

Efficient DMA is critical for high performance of level 1 and level 2 routines since they are memory bandwidth bound. Even though level 3 routines are computation bound, the blocks of the matrices are fetched multiple times. Therefore, unless careful attention is given to DMA related aspects, especially alignment related issues, there can be significant performance degradation in the form of creation of DMA lists, packing/unpacking of data in the SPE local store, etc. We discuss some of the DMA related optimizations for BLAS in this section.

Pre-Processing of Input Matrices for BLAS Level 3 Routines: Pre-processing such as data layout transformation, padding, etc. of the input data [8,14,17] is useful in improving the efficiency of the underlying DMA operations of level 3 routines on the Cell. We rearrange the column-major input matrices into block-layout form, using block size of 64×64, before performing the operation, so that the columns of a block are stored contiguously starting at 128-byte aligned

[1] A set of blocks along the column of the matrix; columnset i refers to the set of all the i^{th} blocks in each row of the matrix.

addresses. This pre-processing is done within the scope of current BLAS routine only. The reformatted matrices are discarded upon completion of the routine. The advantages of pre-processing are:

– *Transfer of Blocks Using Direct DMA*: The block columns are not contiguous in memory, and therefore fetching the blocks requires a DMA list of 64 elements where each list element transfers 64 matrix elements. This DMA list has to be created every time a block is transferred between main memory and local store. When a column does not begin at a 128-byte aligned address, this can lead to significant bandwidth loss. Though this may not impact performance when few SPEs are in service, it can significantly deteriorate performance when there are 16 SPEs – pushing the memory bandwidth to its limits. With pre-processing, each block can be fetched using direct DMA.
– *Reduction in the Number of SPE Kernels*: Several transformations can be applied to input matrices during the pre-processing phase itself. These transformations enhance productivity by reducing the number of different kernels required for different combinations of input parameters such as transpose, triangularity (upper or lower), side (left or right), unit or non-unit triangular, etc. For example, the GEMM operation $C = \alpha A^T B + \beta C$ can be performed using the same kernel as the one used for $C = \alpha AB + \beta C$ by simply transposing the matrix A during the pre-processing phase. For the DTRSM routine, we implemented only 2 kernels to cater to 8 different input parameter combinations by applying such transformations. Similar reductions in kernel implementations were achieved for other routines.
– *Simpler and More Efficient SPE Kernels*: The computational kernels on the SPEs are designed to handle matrix blocks which are properly aligned in the local store. This leads to design of simpler kernels that make effective use of the SIMD features of the SPEs without having to realign the vectors/ matrices based on their current alignment offsets. In the absence of pre-processing, either the vector/ matrix blocks would have to be realigned in memory before invoking the SPE kernels, leading to performance degradation, or more complex SPE kernels would have to be designed.
– *Reduction in Computation within SPE Kernels*: Level 3 routines typically involve scaling of the input matrices. This scaling is carried out in the pre-processing stage itself. This eliminates the requirement of scaling being carried out by the SPE kernel thereby reducing its computation.
– *Reduction in Page Faults and Translation Lookaside Buffer (TLB) Misses*: In the absence of pre-processing, adjacent columns may be in different pages when smaller page sizes are used. Pre-processing can potentially reduce TLB misses under such circumstances [14].

We do not reformat the output matrices. This is because blocks of these matrices are typically updated only once (or few times in some cases) after a large number of computations. Therefore, the cost of fetching blocks of these matrices and reformatting them on the SPEs is fairly small and does not lead to significant performance loss.

Clearly, the suggested pre-processing techniques are feasible only for level 3 routines as level 2 routines have complexity comparable to the memory bandwidth requirements for fetching/ storing the matrix. However, it is feasible to similarly pre-process the vectors in case of level 2 routines. For instance, a strided vector can be pre-processed and copied into contiguous locations so that parts of the vectors can be fetched using direct DMA instead of DMA lists.

All the pre-processing is carried out using SPEs since the SPEs together can attain better aggregate memory bandwidth compared to the PPE. The reformatting of the matrix blocks is independent and therefore lends itself naturally to parallel operations.

Use of Pool of Buffers for Double Buffering: For double buffering statically assigning buffers for all the matrices may not leave enough space in the SPE local store for code and other data structures. However not all the buffers are required at all times. Therefore, in our optimization strategy, we use a pool of buffers from which buffers are fetched and returned back as and when required.

Reuse of DMA Lists: When DMA lists are used for data transfers, creation of the lists is an additional overhead. In the case of I/O data, lists are created both while fetching and storing the data. In our implementation, we minimize the overhead of creating the lists by retaining the list created while fetching the data and reusing it while storing it back.

3.3 Two-Kernel Approach for Level 3 Routines

Highly optimized and specialized SPE kernels are a key component of high performance BLAS routines, especially level 3 routines. We adopt a two-kernel strategy where a set of two kernels is developed for each required combination – a *64×64 kernel* (kernels optimized for blocks of 64×64 elements) and a generic kernel which can process blocks of any dimension which is a multiple of 16 elements and is 16-byte aligned. As mentioned in Section 3.1, the matrices are partitioned into blocks of 64×64 elements and typically a matrix would have a larger fraction of 64×64 blocks as compared to border blocks which may not be of dimension 64×64. If the dimension of a matrix is not a multiple of 64, zeros are padded along that dimension to make it a multiple of 16. This approach limits the maximum number of padded rows or columns to 15 in the worst case and at the same time ensures that the performance of the generic kernel is acceptable because it can still perform SIMD operations. The generic kernels typically show a degradation of less than 10% in comparison to the *64×64 kernel* performance, as shown in Fig. 1(a).

The use of two-kernel approach places significant demand on the memory requirements in the SPE local store. This is also the case when kernels such as GEMM are reused for performing other level 3 operations. However, not all the kernels are required at all times. We use SPE overlays [10] to share the same region of local store memory across multiple kernels. Since one kernel routine is used for most computations (e.g. GEMM in level 3 routines), the amortized overheads of dynamic code reloading are small.

3.4 Efficient Use of Memory

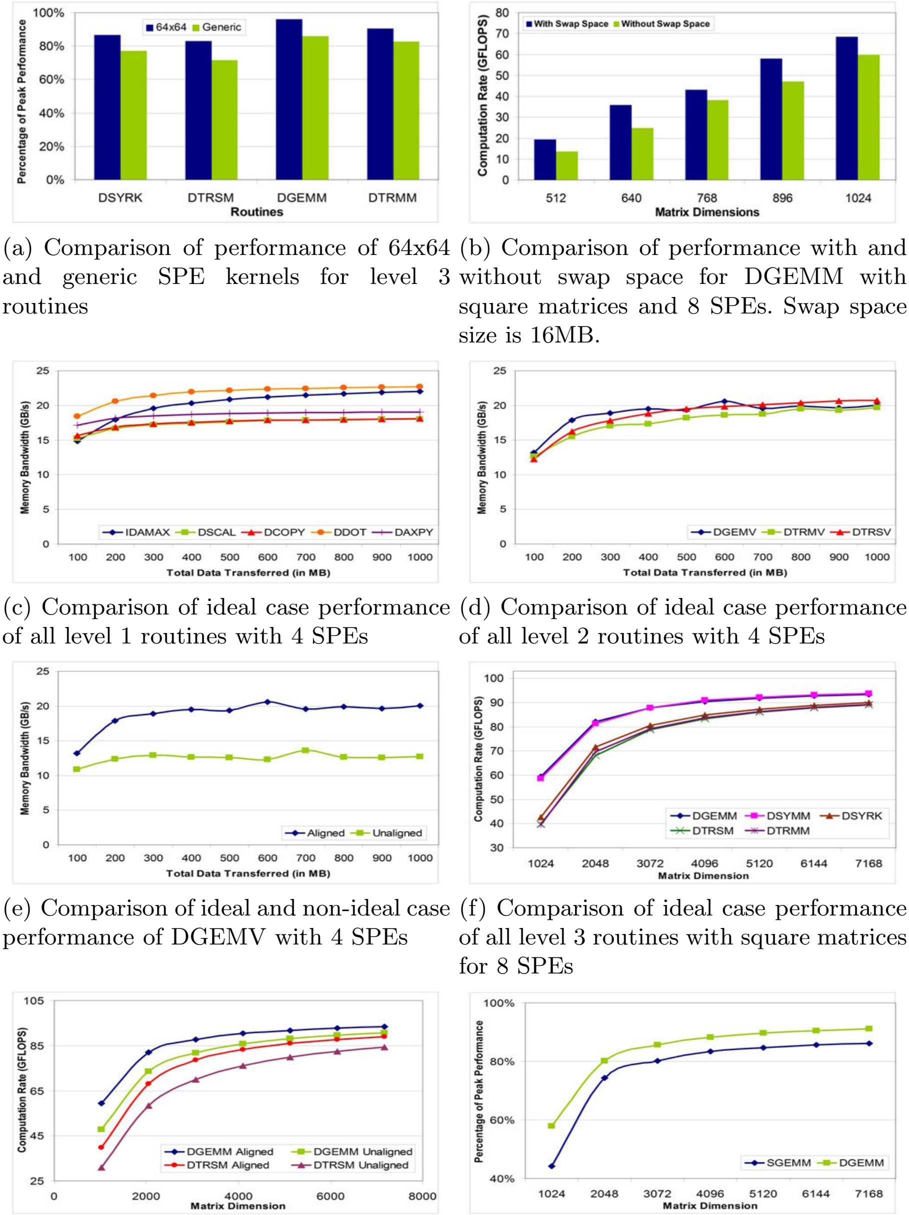
BLAS routines allocate memory internally for pre-processing and rearranging input matrices/ vectors at runtime. There are overheads associated with allocation of memory and accessing it for the first time due to page faults and TLB misses. To minimize this overhead across multiple BLAS calls, a small portion of memory, called *swap space*, can be allocated by the user (using environment variables) and retained across multiple calls of the BLAS routines. The *swap space* is allocated using huge pages. If the internal memory required by the BLAS routine is less than the size of the *swap space*, the routine uses the *swap space* else it allocates fresh memory. This leads to considerable improvement in the performance of the BLAS routines when the input data size is small, as shown for DGEMM in Fig. 1(b).

4 Performance Results

In this section, we report the performance of the BLAS routines obtained with our optimizations. The performance is profiled on IBM Cell Blade (QS22 with 8 GB RAM, Fedora 7, Cell SDK 3.0) with enhanced Double Precision pipeline using GCC 32-bit compiler. Huge pages are used by default. For level 1 and 2 routines, the performance is reported in units of GigaBytes per second (GB/s) since they are memory bandwidth bound and for level 3 routines the performance is reported in units of GigaFlops (GFLOPS).

Figure 1(c) shows the performance results for level 1 routines – IDAMAX, DSCAL, DCOPY, DDOT and DAXPY for ideal input data combinations (i.e. when the starting addresses are 128-byte aligned, stride is 1, dimensions are an exact multiple of their block sizes). We achieve performance in the range of 70–85% of the peak performance (25.6 GB/s) depending on the routine – routines that largely perform unidirectional transfers (e.g. IDAMAX, DDOT) are observed to perform better than the routines that perform transfers in both directions. For level 1 routines, the performance for non-ideal cases, e.g. when vectors are not 128-byte aligned, is almost the same and hence not reported.

Figure 1(d) compares the performance of level 2 routines – DGEMV, DTRMV and DTRSV for ideal input cases (i.e. when data is 128-byte aligned, dimensions are an exact multiple of their block sizes and vector strides are 1). We achieve performance in the range of 75–80% of the peak performance (25.6 GB/s) for level 2 routines. Performance for non-ideal cases (i.e., when data is not properly aligned, leading dimensions are not suitable multiples and vector strides are not 1) is expected to be worse for level 2 routines. Figure 1(e) compares the performance of the DGEMV routine for ideal and non-ideal cases. Performance degrades by about 30% for the unaligned cases. As these routines are memory bandwidth-bound, it is not possible to pre-process the matrix for efficient DMA for unaligned matrices.



(a) Comparison of performance of 64x64 and generic SPE kernels for level 3 routines

(b) Comparison of performance with and without swap space for DGEMM with square matrices and 8 SPEs. Swap space size is 16MB.

(c) Comparison of ideal case performance of all level 1 routines with 4 SPEs

(d) Comparison of ideal case performance of all level 2 routines with 4 SPEs

(e) Comparison of ideal and non-ideal case performance of DGEMV with 4 SPEs

(f) Comparison of ideal case performance of all level 3 routines with square matrices for 8 SPEs

(g) Comparison of ideal and non-ideal case performance of DGEMM and DTRSM with square matrices for 8 SPEs

(h) Comparison of ideal case performance of SGEMM and DGEMM with square matrices for 8 SPEs

Fig. 1. Performance Results

For level 1 and level 2 routines, performance is reported using 4 SPEs. This is because, typically 4 SPEs are enough to exhaust the memory bandwidth and we do not observe significant performance improvement using more SPEs.

Figure 1(f) shows the performance results of level 3 routines – DGEMM, DSYMM, DSYRK, DTRSM and DTRMM for ideal input combinations (i.e. when matrix starting addresses are 128-byte aligned and dimensions are multiples of 64). We achieve up to 80–90% of the peak performance (102.4 GFLOPS). Figure 1(g) compares the performance of DGEMM and DTRSM routines for ideal and non-ideal cases. For the non-ideal cases, the leading dimension is made not to be a multiple of 128 bytes. The performance difference for the non-ideal case is mostly within 10% of the ideal case, demonstrating to a large extent that the pre-processing restricts the performance loss for the non-ideal cases.

We performed additional experiments to determine the performance impact of pre-processing. We found that for ideal cases (described above), the performance with and without pre-processing is comparable, whereas for non-ideal cases, performance degrades by more than 20% when the matrices are not pre-processed. The drop in performance is attributed to the pre-processing required in aligning the fetched blocks and/ or performing matrix related operation (e.g. transpose) before invoking the SPE kernel, and the overhead associated in using DMA lists.

In Fig. 1(h), we compare the performance of SGEMM and DGEMM for ideal input combinations to give an idea of the difference in the performance of the single and double precision routines. It is observed that the performance of the single precision routines shows trends similar to the double precision routines.

5 Conclusions and Future Work

We have discussed the strategies used for optimizing and implementing the BLAS library on the Cell. Our experimental results for double precision show that the performance of level 1 routines is up to 70–85% of the theoretical peak (25.6 GB/s) for both ideal and non-ideal input combinations. The performance of level 2 routines is up to 75–80% of the theoretical peak (25.6 GB/s) for ideal input combinations. The performance of level 3 routines is up to 80–90% of the theoretical peak (102.4 GFLOPS) for ideal input combinations with less than 10% degradation in performance for non-ideal input combinations. These results show the effectiveness of our proposed strategies in producing a high performance BLAS library on the Cell.

The BLAS routines discussed in this paper have been optimized for a single Cell processor, large size data sets and huge memory pages. There is scope for optimizing these routines to optimally handle special input cases, normal memory pages and for multi-processor Cell platforms.

Acknowledgements

We thank Mike Perks and Shakti Kapoor for supporting the BLAS project as part of the Cell SDK, Bhavesh D. Bhudhabhatti, Chakrapani Rayadurgam, Jyoti S. Panda, Pradeep K. Gupta, Lokesh K. Bichpuriya, Niraj K. Pandey, Amith R. Tudur, Ajay A. Kumar, and Sheshasayee V. Subramaniam for their implementation efforts, and Dan Brokenshire and Mike Kistler for sharing their Cell expertise and giving valuable suggestions. Finally, we thank Manish Gupta and Ravi Kothari for their guidance and support to the entire team.

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J., Dongarra, J.J., Du Croz, J., Hammarling, S., Greenbaum, A., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. (1999), `http://www.netlib.org/lapack/lug/index.html`
2. Automatically Tuned Linear Algebra Software (ATLAS), `http://math-atlas.sourceforge.net/`
3. Bader, D.A., Agarwal, V.: FFTC: Fastest Fourier Transform on the IBM Cell Broadband Engine. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 172–184. Springer, Heidelberg (2007)
4. Benthin, C., Wald, I., Scherbaum, M., Friedrich, H.: Ray Tracing on the Cell Processor. In: IEEE Symposium on Interactive Ray Tracing, pp. 15–23 (2006)
5. Dongarra, J.J., Bunch, J.R., Moler, C.B., Stewart, G.W.: LINPACK Users' Guide, Society for Industrial and Applied Mathematics, Philadelphia (1979)
6. Gedik, B., Bordawekar, R.R., Yu, P.S.: CellSort: High Performance Sorting on the Cell Processor. In: Intl. Conf. on Very Large Data Bases, pp. 1286–1297 (2007)
7. GotoBLAS, `http://www.tacc.utexas.edu/resources/software/#blas`
8. Gustavson, F.G.: High-Performance Linear Algebra Algorithms using New Generalized Data Structures for Matrices. IBM J. Res. Dev. 47(1), 31–55 (2003)
9. IBM Corporation. Cell Broadband Engine Programming Handbook v 1.1 (2007)
10. IBM Corporation. SDK for Multicore Acceleration v3.0: Prog. Guide (2007)
11. IBM Engineering Scientific Subroutine Library (ESSL), `http://www-03.ibm.com/systems/p/software/essl/index.html`
12. Intel Math Kernel Library (MKL), `http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/index.htm`
13. LAPACK Working Notes, `http://www.netlib.org/lapack/lawns/index.html`
14. Park, N., Hong, B., Prasanna, V.K.: Tiling, Block Data Layout, and Memory Hierarchy Performance. IEEE Trans. of Parallel and Distributed Systems 14(7), 640–654 (2003)
15. Williams, S., Shalf, J., Oliker, L., Husbands, P., Yelick, K.: Dense and Sparse Matrix Operations on the Cell Processor (2005), `http://repositories.cdlib.org/lbnl/LBNL-58253`
16. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The Potential of Cell Processor for Scientific Computing. In: 3rd Conference on Computing Frontiers, pp. 9–20 (2006)
17. Zekri, A.S., Sedukhin, S.G.: Level-3 BLAS and LU Factorization on a Matrix Processor. Info. Processing Society of Japan Digital Courier 4, 151–166 (2008)

Fine Tuning Matrix Multiplications on Multicore

Stéphane Zuckerman, Marc Pérache, and William Jalby

LRC ITACA, University of Versailles and CEA/DAM
{stephane.zuckerman,william.jalby}@prism.uvsq.fr, marc.perache@cea.fr

Abstract. Multicore systems are becoming ubiquituous in scientific computing. As performance libraries are adapted to such systems, the difficulty to extract the best performance out of them is quite high. Indeed, performance libraries such as Intel's MKL, while performing very well on unicore architectures, see their behaviour degrade when used on multicore systems. Moreover, even multicore systems show wide differences among each other (presence of shared caches, memory bandwidth, etc.) We propose a systematic method to improve the parallel execution of matrix multiplication, through the study of the behavior of unicore DGEMM kernels in MKL, as well as various other criteria. We show that our fine-tuning can out-perform Intel's parallel DGEMM of MKL, with performance gains sometimes up to a factor of two.

Keywords: BLAS, multicore, cache coherency.

1 Introduction

Dense linear algebra, being the first of Berkeley's seven dwarfs [1], is an important part of the scientific programmer's toolbox. BLAS (Basic Linear Algebra Subroutines), and in particular its third level, DGEMM (double general matrix multiplication), are widely used, in particular within dense or banded solvers. It is then no surprise that decades have been spent studying and improving this particular set of subroutines. Over time, theoretical complexity has been improved, while at the same time architecture-conscious algorithms for both sequential and parallel computations have emerged (cf for example Cannon's algorithm [2], Fox's algorithms [4], or more recently SRUMMA [7] and [3]).

There are some reservation to be asserted, however. First, numerous papers focused on the square matrix multiplication case, and not the truly general one. This is particularly damaging because for example the block version of the LU decomposition relies heavily on rank-k updates which are products of an $(N \times k)$ matrix by a $(k \times N)$ matrix with k, typically between 10 and 100, being much smaller than N (typically several thousands); [9] studies this matter extensively. Unfortunately, dealing with these rectangular matrices requires specific strategies fairly different from the standard, easier, square case.

Second, most of the algorithms proposed have a fairly high level view of the target architecture and their underlying model is much too coarse to get the best performance – in terms of gigaflops – of the recent architectures. More

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 30–41, 2008.

precisely, most of the practical algorithms relies on matrix blocking and spreading the block computations across the processors. However fine tuning (choosing the right block size) is still mandatory to get peak performance. This fine tuning process is fairly complex because many constraints have to be simultaneously taken into account: uniprocessor/core performance, including both ILP and locality optimization, has to be optimized, coherency traffic/data exchange between cores has to be minimized and finally the overhead of scheduling the block computations must remain low. In particular, a systematic methodology has to be developed to take into account all of these factors which might have major impact on overall performance. It should be noted that the simpler case of optimizing unicore performance of a matrix multiplication requires a fairly complex methodology (relying on experimental architectural characterization, cf. ATLAS[10]) to reach good performance.

In this paper, we try to develop a parallelization strategy for taking into account all of the architectural constraints of recent multicore architectures. Our contributions are twofold. First we experimentally analyze in detail all of the key factors impacting performance on two rather different multicore architectures (Itanium Montecito and Woodcrest). Second, summarizing our experimental study, we propose a parallelization strategy and shows its efficiency with respect to the well known MKL libraries.

This paper is structured as follows: section 2 describes a motivating example showing the difficulty in selecting the right block sizes, as well as our experimental framework. Section 3 presents experimental analysis of various blocking strategies. Section 4 presents our parallelization methodology and compares the resulting codes with Intel's parallel implementation of MKL.

2 Motivating Example

Experimental Setup

All the experiments shown in this paper have been carried out on the following architectures:

- A dual-socket Xeon Woodcrest (5130) board with dual-core processors, 2 GHz CPU (32 GFLOPS 4 cores peak performance), and 533 MHz FSB (i.e. $\approx 8.6\, GB/s$). Each dual-core processor has a 4 MB L2-cache shared by two cores. This machine will be denoted by "x86" in the remaining of this paper.
- A 4-way SMP node, equipped with dual-core Itanium 2 Montecito processors (with HyperThreading Technology deactivated[1]), with 1.6 GHz CPU (51.2 GFLOPS 8 cores peak performance). Each core has a private 12 MB L3-cache, 256 kB L2-cache and 667 MHz FSB (i.e., $\approx 10.6\, GB/s$). This machine will be denoted as "ia64" in the remaining of this paper.

[1] HTT is mainly useful when dealing with I/O-bound programs, much less with compute-bound ones. Limited testing showed no improvement by using HTT in our computations, while increasing risks of cache-thrashing.

ICC v10.0 and MKL v10.0 were used to make our benchmarks. Two versions of MKL were used: MKL Parallel denotes the orginal parallel version provided by Intel, MKL Unicore (or Sequential) refers to the MKL specially tuned for unicore/sequential use. The operating system is Linux in both cases, with a 2.6.18 kernel.

It should be noted that the MKL Sequential was used as a "black box". It is a very high performance library, on both x86 and IA64 architectures. It shows extremely good results on monocore systems. Thus, aside from the parallel strategy we describe in section 4, a fair amount of tiling, copying and so forth is being performed by the MKL sequential functions. For the remaining of the paper, we will compare our parallelized version of DGEMM based on Sequential MKL kernels with MKL Parallel.

All of the arrays are stored following the "row major" organization, as we used C for our programs. Although the original BLAS library is implemented in FORTRAN, all matrices are stored in a unidimensional array. Experiments show no significant differences between row- and column-major storage strategies in the MKL/BLAS library.

Instead of using OpenMP or directly POSIX threads, we used a performant M:N threading library, Microthread, which was developed internally, and served as a basis for MPC's [8] OpenMP runtime. It relies on a fork-join approach as OpenMP does, but allows for more flexibility – for example by permitting us to chose to which processor we want to assign a given sub-DGEMM, while reducing thread handling complexity inherent to classic POSIX threads. Moreover, thread creation and destruction overheads are kept minimal. However, in terms of performance, the gain offered by Microthread over a solution based on OpenMP remains limited: between 5 and 10% when block computations are small and less than 5% when blocks are large. However, on truly small kernels, where the amount of data makes it difficult to find enough ILP per core, the overhead of Microthread becomes too large (just like any OpenMP runtime). Of course, this is a case where parallelizing a task might prove less beneficial than running a sequential job.

Notations/General Principles of Our Parallelization
For the remainder of this paper, we will look at the simplest form of DGEMM, which performs the following task : $C_{N_1,N_3} = A_{N_1,N_2} \times B_{N_2,N_3}$. We denote NB_i the number of blocks resulting from the partitionning of *i-th* dimension.

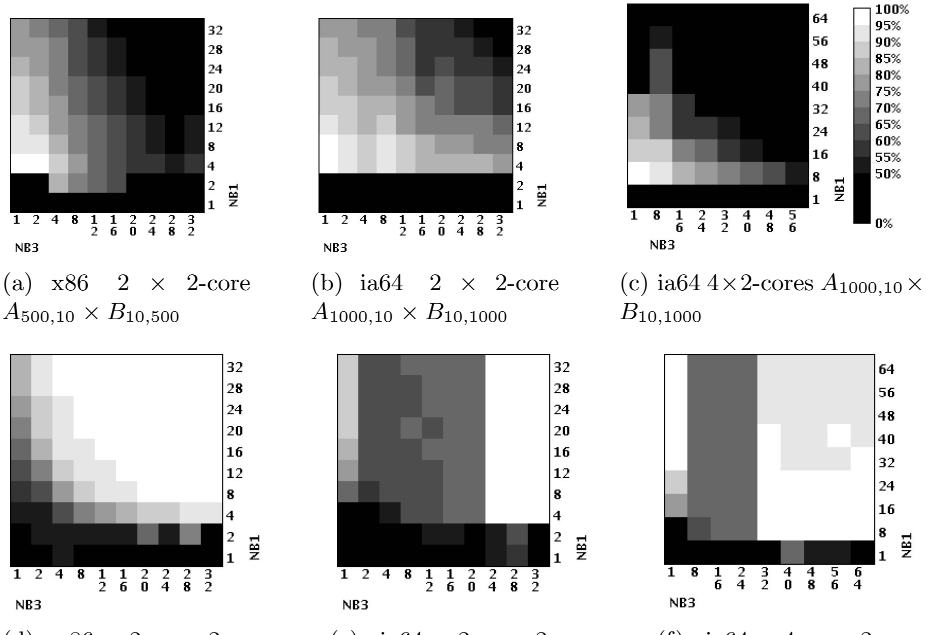
Our parallelization strategy relies on a standard decomposition of the three matrices in blocks ([4, 7]). All of the block computation on a unicore are performed using the MKL library which achieves very good performance on a unicore when the blocks fit in the cache. It should be noted that the blocks resulting from our decomposition are not necessarily square (they can have arbitrary rectangular shape) and second our parallelization strategy is not limited to having a number of block computation exactly equal to the number of available cores. We allow to have much more block computations than cores, i.e. overloading of cores is used.

A Simple Performance Test

Figure 1 describes performance variations of various partitionning strategies for a simple parallel $C_{N,N} = A_{N,10} \times B_{10,N}$: the X axis (resp. Y axis) refers to the number of blocks (NB_3) along the third dimension (resp. the first dimension (NB_1)). Instead of showing absolute performance numbers, relative performance (with respect to the best performance) is displayed: the whiter areas corresponds to best partitionning strategies (i.e. white means between 95 % and 100 % of the best performance), while the darker areas identify poor choices of partitionning parameters.

In the upper three plots (1(a), 1(b), 1(c)) displayed, the size of the matrices are such that they entirely fit in the L2 (resp. L3) cache of the x86 (resp. ia64). Now for these three cases, the white area is much narrower: only one or two partitionning strategies achieve top performance.

In the lower three plots (1(d), 1(e), 1(f)) displayed, the size of the matrices exceed the L2 (resp. L3) cache size of the x86 (resp. ia64). For these three cases, the white areas are fairly large, meaning that many partitionning strategies allow to reach close to the best performance. Now which is much more difficult to predict is the shape of the white area and why the shapes are so different



(a) x86 2 × 2-core $A_{500,10} \times B_{10,500}$

(b) ia64 2 × 2-core $A_{1000,10} \times B_{10,1000}$

(c) ia64 4×2-cores $A_{1000,10} \times B_{10,1000}$

(d) x86 2 × 2-core $A_{2000,10} \times B_{10,2000}$

(e) ia64 2 × 2-core $A_{3000,10} \times B_{10,3000}$

(f) ia64 4 × 2-core $A_{4000,10} \times B_{10,4000}$

Fig. 1. Figure 1(a) (resp. Fig. 1(b), 1(c)): the size of the matrices is such that they fit entirely within the L2 (resp. L3) cache of the x86 (resp. ia64). For figures fig. 1(d) (resp. Fig. 1(e), 1(f)), the size of the matrices is such that they exceed the L2 (resp. L3) cache of the x86 (resp. ia64).

between x86 and ia64. Furthermore, it is a bit surprising that the NB_1 and NB_3 parameters do not have a similar effect on the ia64.

Our Approach
Our goal is to develop a strategy allowing to identify quickly what are good choices for the block values NB_1, NB_2 and NB_3. By "good" we mean within 10% of the best performance achievable when varying arbitrarily block sizes.

To achieve that goal, 3 subproblems have to be carefully taken into account:

1. The block computation running on a unicore must be close to top speed. If the block is too small, there is not enough ILP to get the best performance of the unicore, loop overhead becomes the main reason for slowdowns. If the block exceeds the L2/L3 cache size, the blocking method used by MKL might not be adequate.
2. The number of blocks must be carefully chosen first to achieve a good load balancing and second to keep a low parallelization overhead.
3. The scheduling of block computations to different cores might induce coherency traffic between the cores. For example if a row of C is spread across several cores, each core will write part of the row, some cache lines being shared between two cores (cf Section 3.1).

Finally, it is important to note that we are aiming at the best 10 % as far as performance is concerned, which is symbolized by white or light-grey colors on all our figures.

3 DGEMM Performance Analysis

3.1 Limiting Cache Coherency Traffic

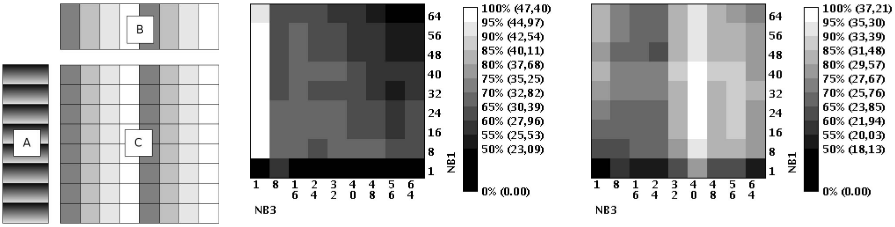
The amount of coherency traffic will depend how blocks are allocated to different cores. We will use two opposite strategies: Write Columns versus Write Rows.

In the Write Columns scheme, every core is computing and writing into different sets of columns of the result matrix C. In this scheme, the A matrix will be read by all cores while each core will read different sets of B columns. Since C is stored row-wise, some cachelines (containing C values) can be shared by different cores leading to coherency traffic. The resulting performance, depending on various blocking strategies are shown in figure 2(b) and 2(c).

In the Write Rows strategy, every core is computing and writing into different sets of rows of the result C matrix. In this scheme, the B matrix will be read by all cores while each will read different sets of rows of the A matrix. In this case, very few cachelines of C are shared between different cores. The resulting performance, depending on various blocking strategies are shown in figure 3(b) and 3(c).

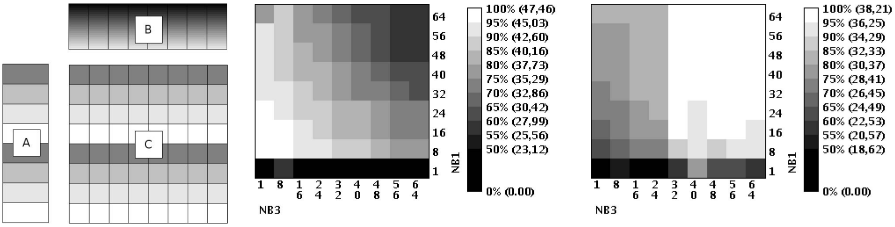
The two strategies are illustrated in figure 2(a) and 3(a).

Although best performance between the two strategies is comparable, one (the Write Columns one) produces a much narrower area of good values for the good block values. On the other hand, the Write Rows strategy gives us an



(a) Write Columns allocation strategy (b) MC 4×2 cores $A_{1000,100} \times B_{100,1000}$ (c) MC 4×2 cores $A_{4000,100} \times B_{100,4000}$

Fig. 2. $A_{N,k} \times B_{k,N}$ blocking with a Write Columns strategy



(a) Write Rows allocation strategy (b) MC 4×2 cores $A_{1000,100} \times B_{100,1000}$ (c) MC 4×2 cores $A_{4000,100} \times B_{100,4000}$

Fig. 3. $A_{N,k} \times B_{k,N}$ blocking with a Write Rows strategy

advantage: the "good" areas encompass the ones in the Write Columns strategy, but are much larger, hence allowing for a bigger blocking factor without hurting performance.

This behavior is clearly due to false-sharing of cachelines: when using the Write Columns strategy, one creates many "frontiers" where a set of cache lines may be shared between two cores. By ensuring that a single core writes for the longest possible time in a same set of rows in C, we reduce these "frontiers" to a minimum. This works because we are in a row-major setup; the strategy would give inverse results in a column-major one.

3.2 DGEMM Analysis

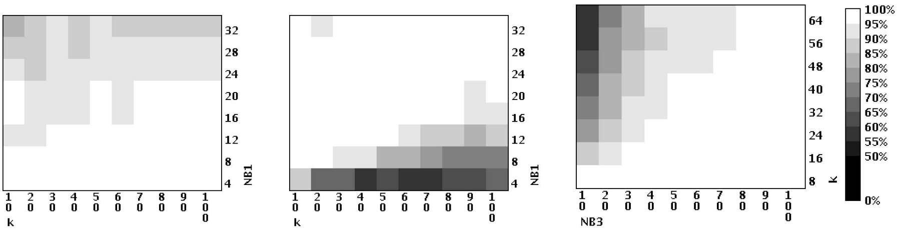
In this section, we will study three extreme cases of matrix multiplication of rectangular matrices, which allows us to uncover most of the key problems in matrix multiplication parallelization. A large set of experiments were carried out. Only the most impressive ones are shown and analyzed. Moreover, we observed a continuous performance behavior when varying parameters such as for example k (where k is the number of columns of A). More precisely, a performance behavior for $k = 20$ can be easily interpolated from the behavior of $k = 10$ and $k = 30$.

Performance counters were not used for this parallel analysis, because efficient tools that give correct and fine measurements in a multicore environment are almost inexistant. You can find good sequential measurement tools such as Perfmon or Intel VTune. Of course, the sequential behavior of a given kernel can help to fine-tune its parallel counterpart (for example, a kernel that already saturates the main memory bandwidth is going to be trouble in parallel). But nothing can be said about cache coherency, and additional bus contention due to several cores trying to write to main memory, for example. However, we do use performance counters while evaluating unicore performance (cf. section 4).

Both the $A_{N,k} \times B_{k,k}$ and $A_{k,k} \times B_{k,N}$ kernels (studied below) behave well in a sequential, unicore environment: performance counters tell us that there is no bandwidth shortage, nor real performance issues.

3.3 Performance Analysis of $C_{N,k} = A_{N,k} \times B_{k,k}$(Fig. 4)

Since k is small, the only opportunity for parallelization lies in partitioning along the first dimension. Each core has its own copy of B, and only relevant rows of A are read. Moreover, writing to C is done row-wise, which prevents most false-sharing from occurring. In figure 4(a) (x86 4 cores) the three matrices fit within the cache and minimizing the partitioning on A i.e. $NB_1 = 4$ or 8 is a fairly good strategy. On the other hand, in fig. 4(b), when we exceed the cache size, larger partitioning degrees of A are required. In figure 4(c) where the three arrays fit again in cache, a minimum number of blocks of A is a very good strategy.

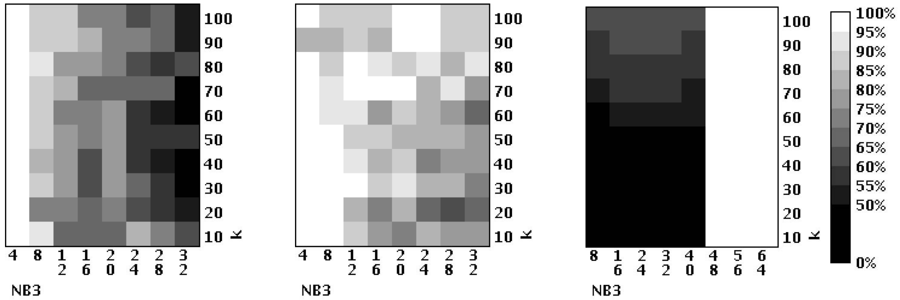


(a) x86 2 × 2-core $A_{2000,k} \times B_{k,k}$ (b) x86 2 × 2-core $A_{6000,k} \times B_{k,k}$ (c) ia64 4×2-core $A_{6000,k} \times B_{k,k}$

Fig. 4. Figures 4(a), 4(b) (resp. 4(c)) present performance variations of the primitive $C_{N,k} = A_{N,k} \times B_{k,k}$ on x86 (resp. ia64). The Y axis refers to the number of horizontal blocks used for partitioning A and B, while the X axis refers to different values of k. For each value of k, performance numbers have been normalized with respect to the best performance number obtained for this value of k. For Figure 4(a) (resp. Fig 4(c)), the size of the matrices is such that they fit entirely within the L2 (resp. L3) cache of the x86 (resp. ia64) while for Fig 4(b), the size of the matrices exceed the L2 x86 cache size.

3.4 Performance Analysis of $C_{k,N} = A_{k,k} \times B_{k,N}$(Fig. 5)

This is the symmetrical counterpart of the previous case. In theory, it should behave exactly the same way, but in practice, there is a huge performance gap. Several factors explain this. The first one is that the performance behaviour of the



(a) x86 2×2-core $A_{k,k} \times B_{k,2000}$

(b) x86 2×2-core $A_{k,k} \times B_{k,6000}$

(c) ia64 4×2-core $A_{k,k} \times B_{k,6000}$

Fig. 5. $C_{k,N} = A_{k,k} \times B_{k,N}$ DGEMMs on x86 (fig. 5(a),5(b)) and ia64 (fig. 5(b)) architectures. Data sets fit in the x86 cache (fig. 5(a)) while data sets in fig. 5(b) exceed its cache size. Fig. 5(c) presents results on ia64 with a data set fitting in L3.

unicore block MKL kernel $B_{k,k} \times C_{k,N}$ is fairly different from the performance of the unicore block kernel $(B_{N,k} \times C_{k,k})$. Second, generating blocks means dividing in a column-wise manner, which is prone to provoke false-sharing.

3.5 Performance Analysis of $C_{N,N} = A_{N,k} \times B_{k,N}$ (Fig. 1)

Here we have a combination between the two previous cases, rendering performance prediction difficult at best. However, there is a clear trend to see: when the sub-matrices fit in cache, there is only one good partitionning strategy, i.e. dividing according to the number of cores. On the contrary, for matrices larger than cache size (see figures 1(d), 1(e) and 1(f)) higher degrees of partitionning are required.

3.6 A Quick Summary of These Experiments

First, basic block performance is essential. Second, as long as we are performing DGEMMs where (sub-)matrices fit in L2 or L3 cache, there is no need to go further than divide the work according to the number of cores available. However, as soon as we are on the verge of getting out of cache, it is important to increase the blocking degree so as to fit in cache once again, with a good sequential computation kernel. So far, all our experiments have shown that this in-cache/out-of-cache strategy (see next section) is sufficient to get good results.

4 A Strategy to Fine-Tune Matrix Multiplication

Methodology for Fine-Tuning DGEMM Parallelization
The major difficulty in the parallelization strategy is in fact the right choice of block sizes (i.e. partitioning of the matrices). Let us first introduce a few

notations. Our focus is the parallelization of the computation of $C_{N_1,N_3} = A_{N_1,N_2} \times B_{N_2,N_3}$. The number of blocks along the first dimension N_1 (resp. N_2, N_3) will be denoted NB_1 (resp. NB_2, NB_3). The corresponding block sizes will be denoted B_1, B_2, B_3, in fact $B_i = N_i/NB_i$, $i \in \{1, 2, 3\}$.

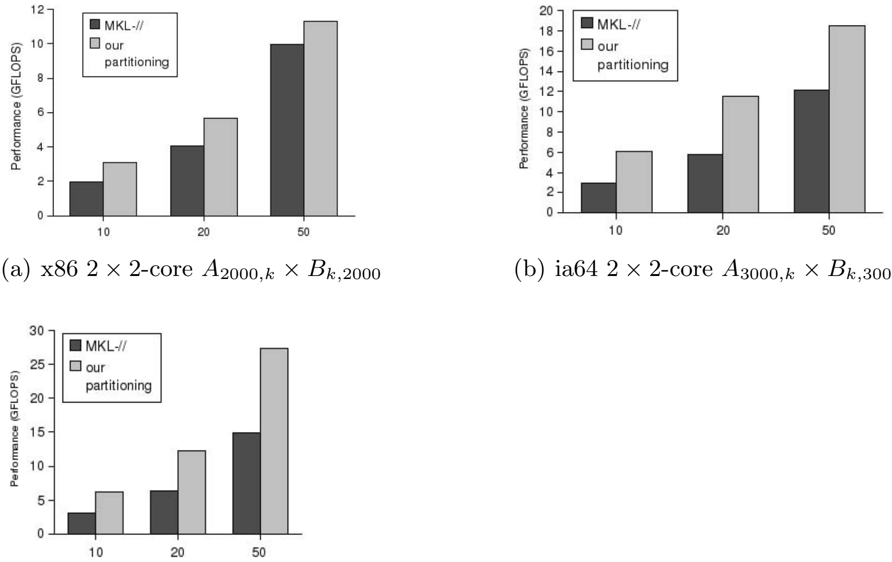
The first step of the method consists in first benchmarking unicore performance of the basic blocks multiplication. This will give us constraints on the block sizes of the form $B_1^{min} < B_1 < B_1^{max}$ (and the similar ones for B_2 and B_3), meaning that if B_1 satisfies such inequalities, we are within 10 % of the peak performance of a unicore matrix multiply. This step requires systematic benchmarking and integrates most of the particularities of the underlying unicore architecture and of the library used for unicore computations. This step is done *once for all* for a given unicore architecture. The results are stored in a database and used in a later step of our strategy. It should be noted that not only GFLOPS performance numbers are stored in this database but also bandwidth consumption between the various cache levels (this is obtained by measuring cache misses using hardware counters).

The second step consists in exhaustively searching all of the partitionings such that:

1. The resulting block sizes satisfy the unicore good performance constraints
2. The sum of the sizes of the three blocks (corresponding to an elementary block computation) is less than a quarter of the last level cache size $B_1B_2 + B_1B_3 + B_2B_3 < CS/4$. Aiming at using only a quarter of the available cache size, allows us to be on the safe side (i.e. being sure that the three blocks remain in cache) and second results still in good cache miss ratio due to the large size of L2 and L3 caches
3. The quantity $NB_1 \times NB_2 \times NB_3$ is a multiple of the number of cores (to insure perfect load balancing). If $NB_1 \times NB_2 \times NB_3$ is less than the number of available cores, the number of cores used is reduced accordingly to still match the load balancing constraint

Then in third step, all of the solutions are lexicographically sorted according to the values of NB_1, NB_2 NB_3. This sort aims at taking into account the fact that from the parallelization point of view the three dimensions are far from being equivalent:

- partitioning along the first dimension induces a simple parallel construct (DOALL type) with minimal overhead and the induced partitioning on matrix C is row-wise and does not induce false-sharing
- partitioning along the third dimension induces also a simple parallel construct with minimal overhead but the induced partitioning on the C matrix is column-wise and will generate false-sharing of cache lines
- partitioning along the second dimension is more complex because it requires synchronization to accumulate the results. In our parallelization strategy, we chose to perform the block computations in parallel, each core accumulating in a different temporary array. Once all of the blocks have been computed, a single core sums up all of the temporary arrays into the final C block.



(a) x86 2 × 2-core $A_{2000,k} \times B_{k,2000}$

(b) ia64 2 × 2-core $A_{3000,k} \times B_{k,3000}$

(c) ia64 4×2-core $A_{4000,k} \times B_{k,4000}$

Fig. 6. Intel's parallel MKL/DGEMM versus our own parallelization

Therefore, the final solution picked up is the one corresponding with the minimum NB_2 value, then the minimum NB_3 value; this corresponds to a lexicographic sort of the solutions. However, in order to minimize cache thrashing, it is important that each thread is given "contiguous" blocks: for each block of lines in A, a given thread which has not reached its maximum number of tasks is given a certain amount of "contiguous" blocks in B.

Very convincing results were obtained using our parallelization strategy (see fig. 6). The most impressive ones relate to the $C_{N,N} = A_{N,k} \times B_{k,N}$ case, where operands do not fit in cache. This is due to the fact that MKL uses a constant strategy of minimizing the number of blocks used (the number of blocks MKL uses is exactly equal to the number of cores). When operands fit in cache, this strategy works fairly well (except in $C_{k,N} = A_{k,k} \times B_{k,N}$) but performs poorly when operands do no longer fit in cache. Although these experiments show how much gain can be obtained with a good parallel strategy, the results are far from reaching peak performance. On the $C_{N,N} = A_{N,k} \times B_{k,N}$ case, there are almost ten times more memory writes than memory reads – i.e., even though there is enough ILP to exploit per core here, writing the results back to memory is tried all at once by all the cores, hence saturating the memory bus.

Comparison with Related Work

ATLAS. ATLAS [10] is a powerful "auto-tuned" library, i.e. upon installation, it performs various measurements (such as determining cache latencies and throughput) in order to choose the best computation kernel adapted to the underlying system. These kernels are either already supplied by expert programmers for a given architecture, or code generated when the underlying system is unknown. ATLAS relies mainly on a good blocking strategy which mixes hand-tuned kernels as well as automatically-generated code at install-time to produce a highly optimized BLAS library. It can also be built into multithreaded library. However, first the number of cores thus supported is fixed, and can never be increased at run-time: one must recompile the whole library each time the number of cores change. Second, ATLAS cannot take easily advantage of already existing DGEMM libraries: it requires very specific kernels.

GotoBLAS. On the opposite side, the GotoBLAS [5, 6] provide a highly hand-tuned BLAS library, with computation kernels programmed directly in assembly language, and very efficient sequential performance as a result. However, these kernels work only on very specific systems (those for which the kernels exist), and do not exactly respect the BLAS semantics (contrary to ATLAS and Intel MKL). Thus, although the changes to one's code are minimal, one can not simply "swap" BLAS libraries with GotoBLAS.

Our approach. It differs from ATLAS and GotoBLAS in different ways. ATLAS and GotoBLAS are above all a work to take advantage of sequential performance. They provide hand-tuned and automatically-tuned BLAS libraries, with an emphasis on blocking. Our approach aims parallel performance only, relying on good sequential BLAS routines. More precisely, our blocking strategy focuses only on parallel performance, with parallel criteria in mind, i.e. sequential ones, as well as memory contention, false-sharing risks, etc. We could take the kernels provided by ATLAS or (with some code modifications) GotoBLAS. Although ATLAS does provides a way to get multi-thread BLAS, this number must be fixed at compile-time, while our method scales with the number of cores.

5 Conclusion

Although matrix multiplication seems to be a solved problem at first, it is clear that in the parallel case and for shared memory systems, a large amount of work remains to be done to get peak performance. It is not enough to use a good and efficient unicore library. Special care has to be taken to take into account behavior of such libraries which are far from being uniform when varying matrix sizes. To get the best out of the MKL in our case, it was necessary to make various trade-offs between data locality, false-sharing avoidance, load-balancing, sequential kernel selection (to get the best sub-DGEMMs cases when distributing tasks) and memory bus contention. This has enabled us to get as much as twice the performance offered by the MKL parallelized by Intel in the best case, in a systematic manner. The methodology we propose is fairly systematic and can be easily automated. However it should be noted that for some specific (small) matrix sizes, the performance obtained is far from peak, due probably to a lack of performance of a unicore version. Further work include improving such cases by generating better unicore kernels then developing a fully automated version of the library and dealing with ccNUMA aspects for larger multicore systems.

References

[1] Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, Univ. of California, Berkeley (December 2006)

[2] Cannon, L.E.: A cellular computer to implement the kalman filter algorithm. Ph.D thesis (1969)

[3] Chan, E., Quintana-Orti, E.S., Quintana-Orti, G., van de Geijn, R.: Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In: SPAA 2007: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, pp. 116–125. ACM, New York (2007)

[4] Fox, G.C., Furmanski, W., Walker, D.W.: Optimal matrix algorithms on homogeneous hypercubes. In: Proceedings of the 3rd conference on Hypercube concurrent computers and applications. ACM, New York (1988)

[5] Goto, K., van de Geijn, R.: High performance implementation of the level-3. Transactions on Mathematical Software 35(1) (2008)

[6] Goto, K., van de Geijn, R.A.: Anatomy of a high-performance matrix multiplication. Transactions on Mathematical Software 34(3) (2008)

[7] Krishnan, M., Nieplocha, J.: Srumma: A matrix multiplication algorithm suitable for clusters and scalable shared memory systems. In: IPDPS (2004)

[8] Marc Pérache, H.J., Namyst, R.: Mpc: a unified parallel runtime for clusters of numa machines. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 78–88. Springer, Heidelberg (2008)

[9] Matthias Christen, O.S., Burkhart, H.: Graphical processing units as co-processors for hardware-oriented numerical solvers. In: Workshop PARS 2007 (2006)

[10] Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. In: Parallel Computing (2001)

The Design and Architecture of MAQAOAdvisor: A Live Tuning Guide

Lamia Djoudi, Jose Noudohouenou, and William Jalby

Université de Versailles, France
lamia.djoudi@prism.uvsq.fr

Abstract. Program performance is tightly linked to the assembly code, this is even more emphasized on EPIC architectures. Assessing precisely quality of compiled code is essential to deliver high performance. The most important step is to build a comprehensive summary for end-user and extract manageable information. In this paper, we present our first prototype called MAQAOAdvisor, a key MAQAO(Modular Assembly Quality Optimizer) module that drives the optimization process through assembly code analysis and performance evaluation. It performs comprehensive profiling, hot-loop and hot-spot detection, fast evaluation and guides local optimizations. An originality of *MAQAOAdvisor* is to deport part of optimizations from the driver to a post-compiler evaluation stage. It is based on static analysis and dynamic profile of assembly code. It feeds information back to help end-user detect and understand performance problems. It proposes optimization recommendations to guide a user to perform the best transformations to get the best performance.

1 Introduction

The quest for performance leads to an ever increasing processor complexity. Similarly compilers are following the same trend with deeper optimization chains involving numerous sets of techniques. As a result code performance is becoming more and more complex to guarantee, it is sensitive to butterfly effects and difficult to assess without extensive tuning and experiments. The Three fundamental points for code optimization are to detect, understand and fix potential performance problems. Nowadays this issue is mostly tackled by using hardware counters and dynamic profiling. An array of tools is used to handle these three stages of performance tuning. Consequently, tuning is a time consuming task, burdensome with a poor productivity. Therefore, a modern approach is much needed, to address the complexity of the task in order to support the multidimensional aspect of performance and complemented existing methods.

We propose an approach which allows us to find the best orientation to guide a user to perform the best transformations to get the best performance. Understanding of how and why the compiler bottleneck occurs, through the feed-back of more information, helps us to execute the code much faster.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 42–56, 2008.

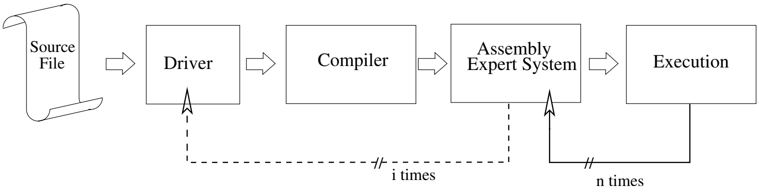


Fig. 1. Adding a stage is a way to cut through the cost of evaluation (by preventing useless execution) as well as to limit the number of evaluations (by preventing the iterative process to apply useless optimizations)

The novelty of our approach is:

1-The optimization part: Which is transferred from the driver to a post-compiler evaluation stage. Being after the compilation phase allows us a precise diagnostic of compiler optimization successes and/or failures, or if due to some obscure compiler decision, the resulting code contains under-performing patterns. Assembly level is the natural place to observe performance, because it is close enough to the hardware and it is possible to check the job done by the compiler. The idea is to enrich the performance ecosystem with a new actor in a collaborative way with the compiler.

2-The assembly code analysis: Our approach gives the first decision about the code quality and which transformation should be applied to improve the quality of assembly code and by consequence it's performance. The use of both static analysis and dynamic profiling within a single framework seems to provide a great amount of flexibility for designers to try out new optimization patterns. By combining static and dynamic analysis, we centralize all low level performance and build correlations.

3-The modification of iterative compilation process: As depicted in Figure 1, our system includes an extra stage between the compiler and the execution. Our approach is located between a model-driven optimization and with machine learning optimization without training. The advantage of this method is to have less N executions than the original iterative compilation so, we speed-up the execution time of the search engine. The driver keeps track of the different transformations to apply next. It reads a list of transformations that it needs to examine together with the range of their parameters. With the original approach, we have only the feedback with the execution time or a few hardware counters. In our approach, we can have more detailed information on the assembly code with an expert system which is in charge of collecting information from an inner-view perspective in contrast with execution time or hardware counters which provide an outer-view. Furthermore, the feed-back provided to the compiler is richer than simple raw cycle counts. This feed-back contains the set of pre-selected transformations than the expert system supposed to be relevant.

In this paper, we present our first prototype called MAQAOAdvisor which is used to provide a live tuning guide capable of improving performance and/or code quality that is not caught by existing tools. It aims to simplify the understanding of the compiler optimizations. To answer the question: is it possible to learn a decision rule that select the parameters involved in loop (application) optimization efficiency ?. We build a summary that defines an abstract representation of loops(application) in order to capture the parameters influencing performance.

MAQAOAdvisor advocates a new approach which can be combined with traditional iterative compilation. This module is characterized by a finer granularity and a richer feed-back. It alleviates the cost of iterative compilation and enlarges the spectrum of candidate codes for optimization.

MAQAOAdvisor, a key MAQAO module drives the optimization process through assembly code analysis and performance evaluation. It is fully implemented in MAQAO (information is presented to the user in a hierarchical manner in a GUI application) MAQAO[1] is a tool which allows the analysis, the manipulation and the optimization of assembly code generated by the compiler. MAQAO tries to identify the optimizations done (or not) by the compiler. Developing *MAQAOAdvisor* as an expert system seems to be a suitable answer as the other generic methods that are not adapted to the highly specific problem of code optimization. It implements a set of rules to help end-user to detect and understand performance problems and make optimization recommendations to guide a user to perform the best transformations to get the best performance.

This paper is organized as follows: Section 2 details MAQAOAdvisor overall design. Section 3 illustrates MAQAOAdvisor outputs. Section 4 details the guided optimization. Section 5 presents related work. And we conclude in Section 6.

2 Overall Design of MAQAOAdvisor

Gathering data and statistics is necessary for a performance tool, but it remains only a preliminary stage. The most important step is to build a comprehensive summary for end-user and extract manageable information. *MAQAOAdvisor* acts as an expert system to drive user attention within the performance landscape. Providing an expert system to help the user to deal with complex architecture was done by CRAY's AutoTasking Expert [2]. It was focused on parallelization issue and was neither as extensible nor as sophisticated as MAQAO's performance module. *MAQAOAdvisor* is built over a set of rules and metrics:

2.1 Performance Rules

Relying on static as well as dynamic information, MAQAOAdvisor implements a set of rules to help end-user to detect and understand performance problem. All rules are written with the support of MAQAO API which allows manipulating MAQAO internal program representation and quickly writing compact rules. Rules can be sorted in three categories:

Transformations Rules: Once assembly code parsing data are stored, the application of the transformations rules will format and gather them according to some conditions in a data table. We detailed four transformations rules:

Issue cost per iteration, jointly with cycle cost, this metric allows to evaluate the cost of data dependences for the loop. A large gap induced by data dependency hints that the loop should be unrolled more aggressively or targeted by other techniques to increase the available parallelism.

Cycle cost per iteration, is expressed as a function of the number of iterations, for non-pipelined loop it is simply in the form of: $a \times N$ where N is the number of iterations. This static cycle evaluation is a reference point to estimate the effectiveness of dynamic performance.

Theoretical cycle bounds per iteration, estimate the data dependency weight in the critical path. This metric[3] indicates if the loop is computationally or memory-wise bound. Knowing whether a loop is computationally or memory-wise bound is a powerful indicator of the kind of optimization techniques to use. Typically computationally bound loops imply that lots of cycles are available to tolerate memory latency problems.

Pipeline loop, where the cost function is: $a \times N + b$. N being the number of iterations, a the cost per iteration and b the filling-up/draining pipeline cost.

Deduction Rules: From the data table of transformation rules, other rules are deduced to help the end-user to (1) detect and understand the performance problem, (2) search effective optimizations, (3) understand optimization failures and obscure compiler decision and to propose code transformations. The deduction rules can be sorted in three categories:

High Level Rules, add semantic to assembly code loop structures. Based on heuristic they are able to compute unrolling factor, degree of versioning, inlining, presence of tail code and report suspicious pipeline depth. These rules also evaluate cost of data dependencies, compute the gap with bound of optimality or hint for vectorization opportunities. Some rules are also dedicated to estimate the purpose of loop versioning. The main cost of loop versioning is the introduction of (a limited) decision tree overhead to select the relevant version, and code size expansion. Several optimizations bring an improvement large enough to overcome this additional cost, but when the gain is questionable, versioning should be turned off.

Code Pattern Rules, are dedicated to rules based on known bad code patterns. For instance on Itanium 2, in some cases the couple of `setf/getf` instructions are used to convert values from the general purpose to the floating point register file. These conversions are costly and in some cases available. Therefore it is valuable to report presence of such patterns. Additionally some rules based on pattern matching evaluate if loops are performing `memcpy` or `memset`. With MAQAOPROFILE, we can have the number of iterations. Also, in MAQAO, we have a summary about some specific functions. For example, the insert of `memcpy` is interesting when the number of iterations is greater than 1000. In this case, a message is reported advising to modify the source code and insert a library call. Additionally `spill/fill` operations are detected, as well as memory operations prone to bank conflicts. On the source level, MAQAOAdvisor also detects if a code is badly written and proposes some high level transformations.

Low Level Rules, address performance problems due to some architecture specifications. For instance in Itanium architecture, it can be: *branch buffer saturation* with 1 cycle long loop body (i.e. one branch to process every cycle). Hardware can not sustains the branch throughput and this leads to some extra stall cycles of the pipeline. Other architecture specific problems like register pressure, or lack of prefetch instruction in a loop with memory operations, and so on.

Additional Rules: MAQAOAdvisor is a library of high level rules which can be extended according to user needs. Users can easily extend the MAQAOAdvisor by writing their own rules.

2.2 Hierarchical Reporting Approach

The needs of the end-user differ, depending on which *level*, the decision is going to be made: is it to chose between two compilers? To select different compilation flags for the whole application? To tune specifically a given loop? Being aware of this, MAQAOAdvisor organizes information hierarchically. Each level of the hierarchy is suitable for a given level of decision to be taken: complete loop characterization, loop performance analysis, function or whole code analysis.

At the first level, the instructions are coalesced per family (e.g. integer arithmetics, load instructions) and counted on a per basic block basis.

The second level, which is already an abstraction layer, only reports loops where some important performance features are detected, thus filters out a large amount of non-essential data. Additionally results are reported in a user-friendly way. This level summarizes the tables of:

(i) selected instruction counts and built-in metrics are displayed which require some knowledge to be interpreted but they represent the exact and complete input of what MAQAO is going to process in the upper stages. However the goal is to detail instructions that have been determined as being of special interest.

(ii) instruction count enriched by built-in metrics : *Cycle cost per iteration*, *issue cost per iteration* and *theoretical cycle bounds per iteration.* Together counts and metrics are exploited by *MAQAOAdvisor rules* which process results gathered during application execution (instrumentation, hardware counters, cycle counts).

(iii) versioning summary for each hot loop. The idea is to perform a study of different versions based on the number of iterations, to decide which is the best version for each interval of iterations, to classify the versions as function of the number of iterations, and choose for each interval of iterations the best one in order to improve parallelism in the original code or in the new optimized code (very interesting to improve the compositional versioning[4]).

The third and forth level, respectively, for each routine and the whole code, a report counting the number of detected performance issues. Reading these tables is quick and was designed to facilitate comparison.

The fifth level summarizes different optimizations. When *MAQAOAdvisor* orients user to generate different versions of each hot loop, MAQAO has the possibility to perform a global study (static analysis, profiling) for all versions at the same time. This automatic process is the ”mode project” in MAQAO.

The sixth level, gives a comparison between different transformations, i.e. for the same code, compiled with different compiler flags; it is possible to do a paired comparison (graphical or tabular). One can perform this comparison for each level (1 to 4) or generate a comparison report.

The seventh level, performs code comparison, i.e. for the same code, compiled with different compilers; here also it is possible to do a paired comparison.

3 MAQAOAdvisor Outputs

Based on the static and dynamic results at all levels, MAQAOAdvisor sorts functions, loops and projects by their respective weight.

3.1 MAQAOAdvisor Modes

MAQAOAdvisor results are displayed in the MAQAO interface or in a report by using the batch mode. MAQAOAdvisor rules and those written by users, can be applied automatically to a large set of files in batch or interactive mode.

3.2 Static Analysis Results

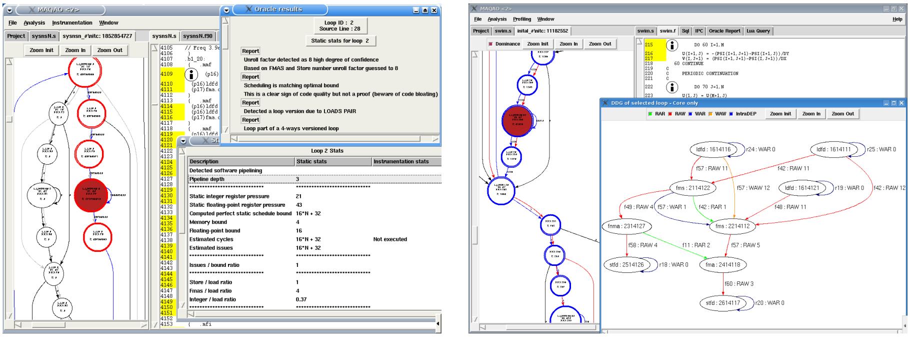
As an analyzer, MAQAO's static module extracts the entire code structure. The structure is expressed through a set of graphs. These graphs are simple yet powerful to analyze a code. Several types of static analysis are also displayed in MAQAOAdvisor. It provides a diagnosis of selected functions, loops or basic blocks like the number of instructions and the information about inner loops.

Call Graph (CG): By selecting one function in CG, MAQAOAdvisor gives all its loops static/dynamic information.

Control Flow Graph (CFG): Represents the predecessor/successor relation among basic blocks and facilitates to display MAQAOAdvisor results for one selected loop (see Figure 2 (a)).

Data Dependency Graph (DDG): Computing the DDG is a key issue to (1) determine critical path latency in a basic block, (2) perform instructions re-scheduling or any code manipulation technique, (3) allow an accurate understanding of dynamic performance hazards, (4) determine the shortest dependency that corresponds to the overlapping bottleneck (see Figure 2 (b)).

Versioning: If the user chooses one loop and click on *versioning button*, MAQAO provides a new window with a summary of the versions of this loop generated by the compiler. If he had performed an instrumentation before, and he clicks on *graph versioning*, *MAQAOAdvisor* provides the distribution of loop iteration count for each version. This information helps us to decide which optimization and version is the best. At this level, *MAQAOAdvisor* can also give a guiding report to do better optimization.



```
4105    // Freq 3.9e
4106    }
4107    .b1_20:
4108    {   .mmf
4109    ⓘ     (p16)
4110     (p16)ldfd
4111     (p17)fma.c
4112    }
4113    {   .mmf
4114     (p16)ldfd
4115     (p16)ldfd
4116     (p17)fma.c
4117    }
4118    {   .mmf
4119     (p16)ldfd
4153    {   .mfi
```

Loop ID : 2
Source Line : 28

Static stats for loop 2

Report
Unroll factor detected as 8 high degree of confidence
Based on FMAS and Store number unroll factor guessed to 8

Report
Scheduling is matching optimal bound
This is a clear sign of code quality but not a proof (beware of code bloating)

Report
Detected a loop version due to LOADS PAIR

Report
Loop part of a 4-ways versioned loop

Loop 2 Stats

Description	Static stats	Instrumentation stats
Detected software pipelining		
Pipeline depth	3	
******************************	******************************	
Static integer register pressure	21	
Static floating-point register pressure	43	
Computed perfect static schedule bound	16*N + 32	
Memory bound	4	
Floating-point bound	16	
Estimated cycles	16*N + 32	Not executed
Estimated issues	16*N + 32	
******************************	******************************	
Issues / bound ratio	1	
******************************	******************************	
Store / load ratio	1	
Fmas / load ratio	4	
Integer / load ratio	0.37	
******************************	******************************	

```
215   ⓘ     DO 60 I=1,M
216           U(I+1,J) = -(PSI(I+1,J+1)-PSI(I+1,J))/DY
217           V(I,J+1) = (PSI(I+1,J+1)-PSI(I,J+1))/DX
218        60 CONTINUE
219       C
220       C   PERIODIC CONTINUATION
221       C
222   ⓘ     DO 70 J=1,N
223           U(1,J) = U(M+1,J)
```

(a) MAQAOAdvisor displays analysis. (b) Data Dependence graph

Fig. 2. SPECFP 2000 benchmark. (a) 178.galgel: close inspection of the loop loop b1_20. In front of each loop of the source code, ⓘ gives access to the information computed by the MAQAOAdvisor concerning this loop. (b) 187.facerec: DDG of hot loop in gaborRoutine. User can choose RAW, RAW, WAR, WAW or intra dependence. It can also visualize them at the same time.

Static Statistics: are the representation of transformations rules detailed in section 2.1 and they can be displayed in the MAQAO interface.

3.3 Dynamic Analysis Results

MAQAO proceeds to code instrumentation automatically[5]. It measures the real application behavior with minimal disturbance. An interesting side effect of our instrumentation is its very low run-time overhead. Profiling information is used to build an execution summary, they can be transparently accessed by end-user or used by MAQAOAdvisor.

3.4 Combining Static/Dynamic Analysis

Example of static/dynamic results:

Prefetch impact: By applying prefetch transformation rules, MAQAOAdvisor detects if a loop containing load or store instructions does not contain prefetch. It warns and advocates for first checking the source code (to consider if data streams are manipulated) and if necessary to use prefetch intrinsics. Intrinsics force the compiler to generate prefetch instructions. This prefetch warning is not emitted in the case of loop tail code, because loop tail codes have only a limited number of iterations. In such a case, the lack of prefetch instruction makes sense.

Value Profiling Results: Time profiling is of limited help for such a fine granularity, but value profiling leads to numerous optimizations. For instance, it is the key metric for code specialization. Additionally, extracting some characteristics of address streams is useful to prevent bank conflicts, aliasing problems or to detect the prefetch distances. Prefetch distances could theoretically be computed off-line with an assembly code analysis. However, it remains easier and safer to rely on dynamic traces, since for instance on Itanium architecture some optimizations allow a single prefetch instruction to retrieve several data streams.

Summary Analysis: By comparing static and dynamic analysis, MAQAOAdvisor detects the value undecidable by a pure static scheme and gives more information to take the best decision.

4 Guided Optimization

MAQAOAdvisor helps end-user to navigate through his code and isolate the particularly important or suspicious pieces of code. For these isolated pieces which are the hot loops, *MAQAOAdvisor* provides as many guidances as possible to help the decision making process. This "guided-profile" allows to understand the compiler optimizations and guides to improve code quality and performance. As detailed in Figure 3, *MAQAOAdvisor* is designed as a set of interlinked levels each of them being loosely coupled to the others. User can take decisions at the end of each level. The best decision is taken at the end of the process.

4.1 Automatic Hot Loops Selection

In this stage, we must find the hot loops to be optimized. MAQAOPROFILE[5] allows us to give a precise weight to all executed loops, therefore underscoring

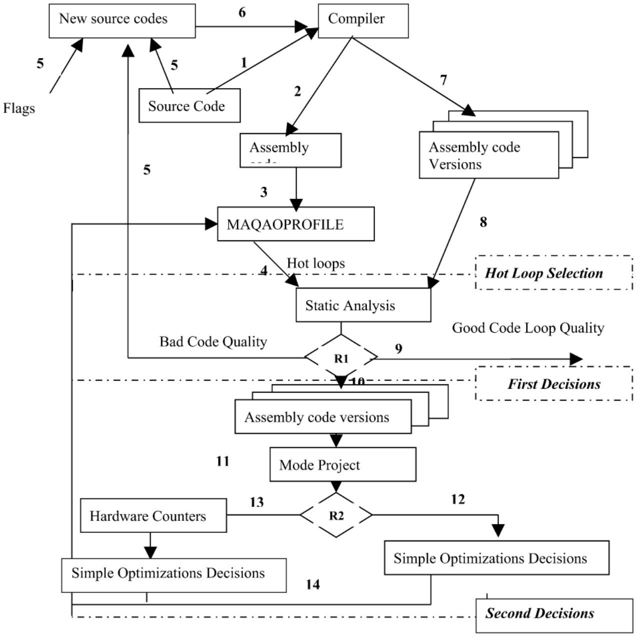


Fig. 3. MAQAOAdvisor Process

hotspots. Correlating this information provides the relevant metrics: (1) Identifying the hotpath at run-time which passes through the whole program where the application spends the most of its time is a key for understanding application behavior. (2) Monitoring trip count is very rewarding, by default most of compiler optimizations target asymptotic performance. Knowing that a loop is subjected to a limited number of iterations allows us to choose the optimizations that characterized by a cold-start cost.

4.2 First Decisions

Based on static analysis, *MAQAOAdvisor* takes first deductions of compiler optimizations and proposes to:

(i) Add "pragma" to avoid (1) the register pressure in order to avoid the spill/fill, (2) the check instructions (that mean compiler had take a bad optimizations), (3) and to inform the user that lot of calls can decrease performance.
(ii) Improve code quality in order to improve performance. Code quality depends of the first ratio $R1$ (issue/bound). It evaluates the matching between static bounds[3] and observed performance. If it is equal to one, then the function/loop is removed from the list of optimization candidates. Otherwise, candidates are evaluated according to several factors: value profiling is used to detect stability.

If we have just one version with $R1 \leq 1.2$ and there is no problem of spill /fill, check instructions and functions calls, *MAQAOAdvisor* decides that is the best one and the process can be stopped here.

If $R1 \geq 1.3$, we generate the first guided optimization. It combines the static and dynamic analysis of the original version of each hot loop. Then it allows user to apply the first optimization for the hot loops in source or assembly level to improve code quality and the performance. For example, it can propose optimization on source level, like software pipelining, unrolling, add prefetch.

At this level, applying different transformations for several hot loops in assembly or source level, implies the generation of several versions of code. The analysis of these versions allows to find the best version or what kind of transformations user must take, to have the best performance at the second level of the *MAQAOAdvisor*. It is possible that the compiler may not improve the code quality, so *MAQAOAdvisor* orients user to the second decisions.

4.3 Second Decisions

Once at this level, we are sure that we can improve the performance more than the previous level. To find the trade-off between quality and performance it is interesting to calculate the second ratio $R2$.

$R2 = \frac{\mathrm{c}_2(N)}{\mathrm{c}_1(N)}$ where: $\mathrm{c}_2(N)$ = number of cycles executed for N iterations.

$\mathrm{c}_1(N) = A_1.N + B_1$, where A_1 is the static cycles of the body, B_1 is static cycles spent in overheads and N is the number of iterations of the loop.

If the compiler unrolls the original loop and generates a remainder loop, the formula of c$_1$ is: $c_1(N) = A_1.N + B_1 + a_1.(N \bmod UF) + b_1$ where a_1 and b_1 are the parameters of the loop corresponding to remainder iterations and $(N \bmod UF)$ is the remainder iterations and UF is the unrolling factor. This ratio answers the question: Does the static code represent a good dynamic behavior?

To take a decision to what we do, *MAQAOAdvisor* combines the information like $R1$ and $R2$ values of one or more versions for each hot loop:

Simple Optimizations decisions: *MAQAOAdvisor* follows this path for the good $R2$ value ($R2 \leq 3$) and decides to guide user to:

Combining best versions in the same source code, where $R1 \leq 1.2$. It is a high level optimization. To achieve a trade-off between code quality and performance, *MAQAOAdvisor* combines for each hot loop and their best versions: (1) the unrolling factors, (2) the loop and code size, (3) $R1$ and $R2$. All this process is automatic. This information is given to a solver that finds the trade-off. *Rescheduling*, where $R1 \geq 1.2$. A generation of the DDG of the loop can help us to reschedule the assembly instructions in order to improve the code quality. We choose the version that corresponds to the small $R2$.

Compositional loop specialization, we can also apply a low level optimization. It's indepent of the $R2$ value and it can complete and give more performance than the two first optimizations. Knowing the number of iterations, this technique[4] can generate and combine sequentially several versions at the assembly level. We can get best performance with this technique because we improve the best versions using the MAQAOAdvisor decisions.

Complex Optimizations decisions: If we have a bad $R2$ value ($R2 \geq 3$), *MAQAOAdvisor* guides user to use hardware counters. An interesting advantage, the hardware counters are implemented in MAQAO. Executing a simple script in MAQAO, *MAQAOAdvisor* combines the hardware counters and MAQAO results to guide user to take a decision. For example to solve the cache misses, *MAQAOAdvisor* can propose one of the decisions:

Memory reuse, by modifying the stride of the loop or aggregating the data.

Optimization cache, taking a copy of data or a blocking cache decrease the TLB.

Recovery of Data access latencies, by adding a pragma in source code or modify the prefetch distance in assembly code. This modification is still in progress in the compositional approach implemented in MAQAO.

4.4 Optimization Results

In this section, we evaluate our proposed technique. We consider three benchmarks: CX3D application, a JACOBI code, and a benchmark from the SPECFP2000.

Experiments were run on a BULL Itanium 2 Novascale system, 1.6GHz, 3MB of L3. On the software side, codes were compiled using Intel ICC/IFORT 9.1.

CX3D: CX3D is an application used to simulate Czochralski crystal growth a method applied in the silicon-wafer production. It covers the convection processes occurring in a rotating cylindrical crucible filled with liquid melt.

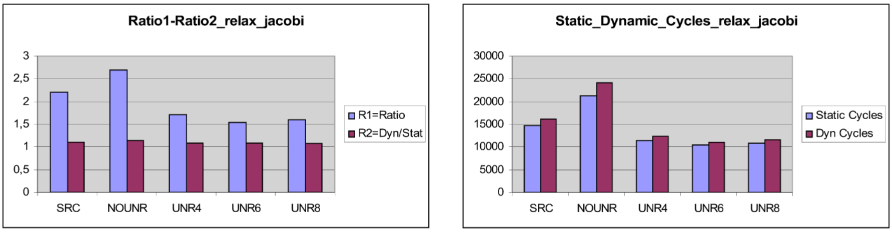
Based on hardware counters technique, we count the cycles, instructions and nop retired as well as back_end_bubble_all stall cycles, we remark that back_end _bubble_all stall cycles are the most important (more than 50%). To know the reason of this stall, we must analyze the subevents. The BE_L1D_FPU_BUBBLE dominates (86.21%). To know the reason of this stall, we must observe two events where BE_L1D_FPU_BUBBLE_L1D takes 99.54%. To have more precision, we observe different sub-events for this event. The cause in this level is that the compiler had a problem to load integer variables from L1D in one cycles (BE_L1D_FPU_BUBBLE_L1D_DCURCIR takes 51%). Arriving at this level, we do not have more precise information and we must take another approach to understand the problem.

But if we use our process, we are sure that we take less time than trying to understand the hardware counters results in order to identify the problem and then give a solution. With our approach, firstly we can just base on static analysis giving the first diagnostic. Combining static and dynamic analysis, our system can give a precise diagnostic and a precise solution to improve performance. For example, for this loop, one of the suggestions is the memory access aliasing. Our aliasing memory module proves that we have an aliasing problem. After that a precise solution proposed by our system is "you must apply an interchange". The process is organized as follows: first a fine grain profiling is done to get accurate hot functions and for the hot functions we give the accurate hot spots. Then the most time consuming inner loops are optimized according to their static and dynamic analyses of our method.

1 - MAQAOPROFILE Information:

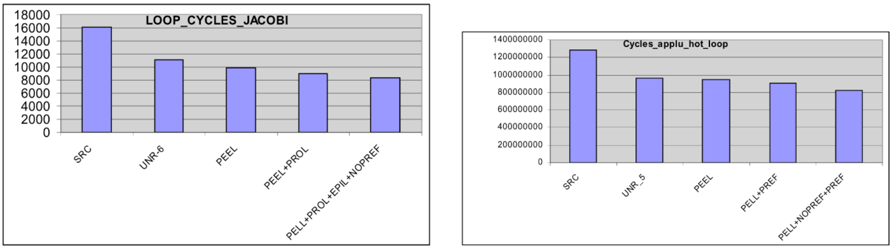
(i) Hot functions: the time attributed to the highest routine (velo) is 70.12 %.
(ii) Hot loops: to isolate the most time consuming loops. The hot loop which is at the source line 787. Other loops have been omitted for sake of clarity.

2 - Optimization sequence: Based on MAQAOAdvisor process, we try to improve code quality in order to improve performance. Our approach is applied to loop id 75. Before applying different transformations and relying only on the combination of static and dynamic information, MAQAOAdvisor (i) collects compiler optimization information applied to this loop , (ii) proposes different solutions (unrolling, prefetching and interchange) for this loop. Generating these versions and the summary of the static information, the GLPK solver indicates that the interchange transformation is better. The gain is 60% and R1 becomes 1 (good code quality). To prove our approach, we have also applied profiling for these transformations. We have remarked that there is a correspondence between solver solution or proposition and dynamic results. That proves, it is not necessary to execute different versions in order to find the best execution time corresponding to the best transformations. But just with the useful static information, we can find the best transformations.



(a) Static guided-profile (b) Dynamic guided-profile

Fig. 4. Hot file (relax_jacobi), hot loop (2655 source line), selected versions: (a) first and second ratio. (b) static cycles and dynamic cycles.



(a) Compositional versioning of jacobi (b) Compositional versioning of Applu

Fig. 5. CPU cycles for different compositional versioning: (a) loop 23 (source line) in relax_jacobi, (b) loop 2655 (source line) of 173.applu

JACOBI: Jacobi code solves the Helmholtz equation on a regular mesh, using an iterative Jacobi method with over-relaxation. The first level of our approach demonstrates that Jacobi contains one important hot loop (source line 2655). This level allows us to generate some versions of this loop using pragma. Introducing all guided-profile important information to the GLPK[10] solver, it finds a trade-off and decides the version unroll 6 is the best one (see Figure 4). For the best version, we have applied different transformations. See Figure 5(a).

173.APPLU: It is a benchmark from SPECFP2000 which leads to the performance evaluation of the solver for five coupled parabolic/elliptic partial differential equations. The same process of jacobi was applied for this benchmark and the best version is the version unroll 6. Accurate results of compositional versioning are provided in Figure 5(b).

5 Related Work

Very few tools focus at providing user with transformation code advices for performance tuning. Tools such as foresys [6] or FORGExplorer [7] propose code analyses as well as code transformations but no techniques to identify the tuning transformation to use. Vtune[12] is mainly a profiling tool. Its usage is so widespread that an API gets standardized to describe their access. CAHT[8] shares the same goal as Vtune: ”discover performance-improvement opportunities often not considered by a compiler, either due to its conservative approach or because it is not up-to-date with the latest processor technology”. CAHT also formalized the search of tuning advices and so builds an easily extensible system based on case-based reasoning (CBR). The solution proposed by CAHT is not precise when there are no similar cases because it must ignore some characteristics to provide a solution. We propose to extend the MAQAOAdvisor to incorporate the case-based reasoning but just for similar case. With each new case and the use of an expert system, we are sure we will enrich the knowledge base with very precise cases. In addition to the combination between the CBR and expert system, we will propose precise solutions.

ATOM [11] and Pin [15] instrument assembly codes (or even binary for Pin) in a way that when specific instructions are executed, they are caught and user defined instrumentation routines are executed. While being very useful Atom and Pin are more oriented toward prospective architecture simulation than code performance analysis. EEL[9] belongs to the same categories of tools. This C++ library allows editing a binary and adding code fragment on edges of disassembled application CFG. Therefore it can be used as a foundation for an analysis tool but does not provide performance analysis by itself. Currently EEL is available on SPARC processors. Vista [13], is an interesting cross-over between compiler and performance tool. Plugged with its own compiler, Vista allows to interactively test and configure compilation phases for code fragment. Everything is done in a very visual way. While being conceptually close to MAQAO, Vista remains more a compiler project than a performance analyzer.

Shark [14] offers a comprehensive interface for performance problems. Like MAQAO, it is located at the assembly level for its analyzes, displays source code as well as profiling information. As most of Apple’s software the GUI is extremely well designed. However Shark lacks instrumentation and value profiling. Code structures are not displayed and the Performance Oracle advices are currently limited to very few messages: alignment, unrolling or altivec (vectorization). Additionally as most of Apple’s software it is very proprietary and does not offer open-source scripting language or standard database. Nevertheless it remains an advanced interface, with an extensive support of dynamic behavior and it underlines the need to think performance software beyond gprof.

6 Conclusion

MAQAO is a tool that centralizes performance information and merges them within a representation of the assembly code. MAQAO also provides several views on the internal representation of an input program that the user can navigate through. MAQAOAdvisor module drives the optimization process by providing support through assembly code analysis and performance evaluation. It explores the possibility to let a user interact with program analysis and opens new ways of exploring, modifying and optimizing assembly and source code.

Taking advantage of precise profiling information, our system is able to select the most suitable optimizations among a list defined by the user (Deep Jam, ...) or using directives (unroll, software pipelining, prefetch,...). A trustable API to drive exactly the sequence of optimizations through the compiler would be useful to unleash the potential of our (any) feed-back approach.

Our goal is to improve MAQAOAdvisor to a real expert system. The idea is to present a prototype which is a design expert system for MAQAO incorporating case-based reasoning. The case-based reasoning is the method in which we create a knowledge base. If you have a new application, the system searches a domain dependent case-base for a similar case:

(i) If there is one, the system uses it to propose solutions to improve performance and/or code quality with minimum user interaction. To do this we must analyze the application and identify its characteristics and its context.
(ii) When there is no similar case, instead ignoring certain characteristics (the case of case-based reasoning), we can leave it to the user or an intelligent system. With each new case and the use of an expert system, we are sure we will enrich the knowledge base with very precise cases. In addition to the combination between the CBR and expert system, we will propose precise solutions.

As a compiler construction tool, our framework can be useful to compare different compilers. For instance, it is easy to track regressions between two versions of a compiler or to have an accurate picture of compilation flag impact.

The main goal is to export MAQAO to different VLIW compilers. The implementation of Trimedia architecture is still in progress. The assembly code generated by Trimedia is very similar to Itanium 2 and we implement an interface in MAQAO in order avoid writing another MAQAO specific to Trimedia.

In the future, MAQAO will include improving optimization module by adding new optimization techniques, based on powerful mathematical models.

References

1. Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.-T., Jalby, W.: MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2. In: Workshop on EPIC, San Jose (2005)
2. Kohn, J., Williams, W.: ATExpert. Journal of Parallel and Distributed Computing 18(2), 205–222 (1993)
3. Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.-T., Jalby, W.: Exploring Application Performance: a New Tool for a Static/Dynamic Approach. In: Los Alamos Computer Science Institute Symposium, Santa Fe, NM (2005)
4. Djoudi, L., Acquaviva, J.-T., Barthou, D.: Compositional Approach applied to Loop Specialization. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 268–279. Springer, Heidelberg (2007)
5. Djoudi, L., Barthou, D., Tomaz, O., Charif-Rubial, A., Acquaviva, J.-T., Jalby, W.: The Design and Architecture of MAQAOPROFILE: an Instrumentation MAQAO Module. In: Workshop on EPIC, San Jose (2007)
6. FORESYS, FORtran Engineering SYStem, http://www.pallas.de/pages/foresys.htm

7. FORGExplorer, http://www.apri.com/
8. Monsifrot, A., Bodin, F.: Computer aided hand tuning (CAHT): applying case-based reasoning to performance tuning. In: ICS 2001: Proceedings of the 15th international conference on Supercomputing (2001)
9. Larus, J.R., Schnaar, E.: EEL: Machine-Independent Executable Editing. In: The ACM SIGPLAN PLDI Conference (appeared, June 1995)
10. http://www.gnu.org/software/glpk
11. Srivastava, A., Eustace, A.: ATOM - A System for Building Customized Program Analysis Tools. In: PLDI 1994, pp. 196–205 (1994)
12. VTune Performance Analyzer, http://www.intel.com/software/products/vtune
13. Zhao, W., Cai, B., Whalley, D., Bailey, M., van Engelen, R., Yuan, X., Hiser, J., Davidson, J., Gallivan, K., Jones, D.: Vista: a system for interactive code improvement. In: Proceedings of the joint conference on Languages, compilers and tools for embeded systems, pp. 155–164 (2002)
14. http://developer.apple.com/tools/shark_optimize.html
15. Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., Karunanidhi, A.: Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation Micro 37, Portland (2004)

A Load Balancing Framework for Clustered Storage Systems

Daniel Kunkle and Jiri Schindler

Northeastern University and NetApp Inc.

Abstract. The load balancing framework for high-performance clustered storage systems presented in this paper provides a general method for reconfiguring a system facing dynamic workload changes. It simultaneously balances load and minimizes the cost of reconfiguration. It can be used for automatic reconfiguration or to present an administrator with a range of (near) optimal reconfiguration options, allowing a tradeoff between load distribution and reconfiguration cost. The framework supports a wide range of measures for load imbalance and reconfiguration cost, as well as several optimization techniques. The effectiveness of this framework is demonstrated by balancing the workload on a NetApp Data ONTAP GX system, a commercial scale-out clustered NFS server implementation. The evaluation scenario considers consolidating two real world systems, with hundreds of users each: a six-node clustered storage system supporting engineering workloads and a legacy system supporting three email severs.

1 Introduction

The basic premise of clustered storage systems is to offer fine-grained incremental capacity expansion and cost-effective management with performance that scales well with the number of clients and workloads [1,2,3,4,5,6]. To address load imbalance, most previously proposed architectures either dynamically redistribute individual data objects and hence load among individual nodes in response to changing workloads [2,5], or use algorithmic approaches for randomized data allocation (e.g., variants of linear hashing [7]) to distribute workload across cluster nodes [4,6].

However, the first approach is not well suited for enterprise storage systems. First, deployed systems typically collect only cumulative statistics over a period of time [8,9], as opposed to detailed traces with per-request timings [10,11]. Yet, systems with data migration at the level of individual objects [2,5] typically use techniques that require detailed traces to make informed decisions [12]. Second, workloads do not always change gradually. They often do so in distinct steps, for example, during consolidation when an existing system inherits a legacy system workload.

A complementary approach to balancing load across system components is to use offline solvers [8,13]. They typically use a variant of the bin-packing or knapsack problem [14] to find a cost-efficient system configuration (solution). They require only a high-level workload description [15] and capacity or performance system model. While such solvers have been shown to be effective for building an enterprise-scale system from the ground up, they are less suitable when already deployed systems grow or experience workload changes over time. Previous work proposed to iteratively apply

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 57–72, 2008.

constraint-based search with bin packing [16]. However, doing so does not take into account the *cost* of the system configuration change and the resulting impact of potential data movement on system performance.

To address the shortcomings of the existing solutions, we have developed a load-balancing framework with two primary objectives: (1) It should be modular and flexible, allowing for a range of definitions for *system load* and *imbalance*, as well as for many types of optimization techniques instead of using one specific algorithm. (2) The cost of reconfiguration should be a primary constraint, guiding which solutions are feasible and preferred. We use a combination of analytical and empirical estimates of cost, along with a measure of system imbalance, to define a multiobjective optimization problem.

Using this framework, we implemented a load balancing system tailored to the specifics of the NetApp Data ONTAP GX cluster [1]. Our approach is grounded in *real* features of our deployed systems. We are motivated to find *practical* solutions to problems experienced by real users of our systems; for example, how to best consolidate existing application workloads and legacy systems into one easier-to-manage cluster. We focus on balancing the load of an already operational system; a scenario more likely to arise in practice than designing a new system from the ground up. We also explore the use of more formal techniques in the context of production enterprise systems. This motivation has been recently echoed by a call to employ optimization methods already in use by the operations-research community in place of more ad-hoc techniques prevalent in the computer systems community [17]. We demonstrate the applicability of our approach using an internally deployed Data ONTAP GX cluster hosting engineering workloads and home directories. We examine a scenario of consolidating storage in a data center—rolling a legacy Data ONTAP 7G system with e-mail server workload into a Data ONTAP GX cluster supporting software development.

2 Load Balancing Framework Overview

The primary goal of our framework is to provide an abstract method for load balancing that is applicable to a wide range of workloads and systems, as well as allowing for many different policies and strategies. To facilitate this, we have divided the framework into four modules, each of which can be modified without requiring significant changes to any of the others. At a high level, the framework represents a canonical decision system with a feedback loop—a model that has previously been shown to work well for storage system configuration [16].

Figure 1 shows the general structure and components of our modular load-balancing framework. The Observe Load module records, stores, and makes available a set of statistics that characterize the load on the system. The Detect Imbalance module calculates the *imbalance factor* of the system—a measure of how evenly the load is distributed. If the imbalance factor passes some threshold, the Optimize Reconfiguration Plan module is invoked. It determines a set of system reconfigurations that will mitigate the load imbalance while minimizing the cost of those reconfigurations. The module Execute Reconfiguration Plan executes a series of system-level operations.

We now describe each module in greater detail. We first give a general definition of the module, followed by the details of how that module is applied to the specifics of the Data ONTAP GX cluster. The system architecture is detailed elsewhere [1].

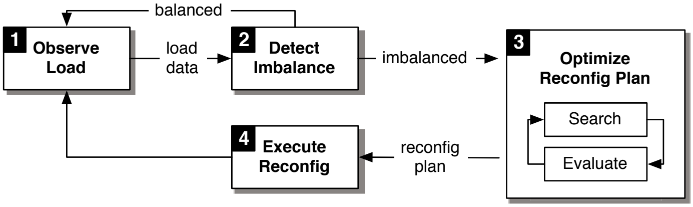


Fig. 1. Flow diagram of the load-balancing framework

2.1 Observe Load

We characterize load with two concepts: *element* and *load component*. An *element* is an object in the storage system that handles load, that is, something that can be *overloaded*. This can be a hardware component (e.g., a controller with CPU and memory), an architectural component, or a software module. A *load component* is an indivisible source of load that can be migrated, or otherwise reconfigured, to alleviate the overloading of an element. This can be a single data object (e.g., a file) or a logical volume.

An important factor for observing load is the frequency of data collection. In practice, this is driven by the constraints of the collection method used, for example, tracing every request versus having an agent periodically sample various counters. Another important factor is the time period over which the load is observed and decisions are made. If that time period is too short, the system may react too quickly, attempting to re-balance after a transitory spike in load. Conversely, if the time period is too long, the system may remain in an imbalanced state for an extended period of time. This period also depends on how long it takes to implement changes. In particular, it should be long enough to allow a new steady state to be reached and evaluated before acting again. Finally, load can be expressed by one or more variables, for example, we can collect both the throughput and latency of I/O operations over some time period.

Application: Load imbalance in the Data ONTAP GX cluster can be caused in two ways. First, nodes (Nblades) can be overloaded by client requests, which they must process and forward. This imbalance can be mitigated by directing client requests away from heavily loaded nodes by moving an existing virtual interface (VIF) from one Nblade to another. Second, a node (Dblade) may be overloaded if the data it contains is heavily requested. This imbalance can be mitigated by migrating data (i.e., volumes with separate file system instances), or by creating load-balancing volume mirrors on other nodes with lighter load. Dblades and Nblades constitute elements, each with their respective load components. We demonstrate their interrelationship with two scenarios.

Scenario 1: Balancing Client-Request Handling. An *element* is a networking component (Nblade), which handles client connections and routes requests to the target data component (Dblade). A *load component* is a virtual interface (VIF), through which clients requests are handled. Each VIF is assigned to a single Nblade. The system can be reconfigured by migrating a VIF from one Nblade to another; the new node will handle all future client requests through that VIF.

Scenario 2: Balancing Data Serving. An *element* is a data component (Dblade), a compute node that receives requests from Nblades and serves the requested data to the client. A *load component* is a volume, a file system subtree, containing the directories and files in that subtree. The system can be reconfigured by migrating a volume from one Dblade to another or mirroring the volume on another Dblade.

For our evaluation in Section 3, we consider the second scenario, defining load in terms of operations per second contributed by each load component (volume).

2.2 Detect Imbalance

Module 2 compresses the multidimensional expression of load to a single value. This *imbalance factor* describes the extent to which the load on the system is evenly distributed. When this value passes some threshold, Module 3 is invoked to produce a set of possible reconfiguration plans. Even though this module reduces the load description to a single value to determine when rebalancing is appropriate, the full load description is still made available to the optimization methods in Module 3.

The imbalance factor computation uses two different functions to (i) compress the temporal load on each element to a single scalar value and (ii) calculate the final imbalance factor from the set of per element load values. We have developed a number of possible functions for each of these steps. By using different functions, one can choose under which conditions rebalancing occurs and which possible reconfigurations are optimal. For purposes of describing these functions, we define the following notation. Let u_t^e be the load on element e at time t. For example, u_t^e could be the number of I/O operations performed through element (node) e over some period of time (e.g. one minute). The values $u_1^e, u_2^e, \dots, u_n^e$ would then define the number of I/O operations served by e over the last n minutes. Let L^e define the load on e with the temporal component removed.

Reduction of Temporal Component. First, we present three possible functions for removing the temporal component from the load observed on each element. Note that the functions are applicable regardless of the frequency of observations (i.e., the granularity of our measurements) or the period over which we apply the given function. The functions are defined in Table 1. The simple sum function adds the load on e at each time, placing equal weight on all possible load values. This is equivalent to a moving average of the load on an element.

The polynomial function emphasizes large utilization values. This can be useful in identifying situations where an element sees a large spike in utilization for a small period of time. With the simple sum function above, this spike would be lost. With this polynomial function, the spike is emphasized and corresponds to a larger increase in

Table 1. Reduction of temporal component functions

Simple Sum	*Polynomial*	*Threshold*
$L^e = \sum_{i=1}^{n} u_i^e$	$L^e = \sum_{i=1}^{n} (u_i^e)^\alpha$	$L^e = \sum_{i=1}^{n} (u_i^e \cdot T(u_i^e, k)); \quad T(u_i^e, k) = \begin{cases} 0 & \text{if } u_i^e < k \\ 1 & \text{if } u_i^e \geq k \end{cases}$

Table 2. Requirements for imbalance function

Definition	*Description*
$0 \leq f(\mathbf{L}) \leq 1$	The range of values is from zero to one.
$f(0,0,\ldots,1) \rightarrow 1$	The maximum value is defined as a single element handling all load. Any load handled by more than one element should have a value less than one.
$f(1/n,1/n,\ldots,1/n) \rightarrow 0$	The minimum value is defined as a perfectly balanced load. Any other load should have a value greater than zero.
$f(a,a+\varepsilon,\ldots) > f\left(a+\frac{\varepsilon}{2},a+\frac{\varepsilon}{2},\ldots\right)$	Moving load from a more loaded element to a less loaded one reduces the value.
$f(\mathbf{L}) < f(0,\mathbf{L})$	Adding a new element with zero load increases the value.

L^e than if that same load had been evenly spread over time. In the polynomial function definition $\alpha > 1$; larger values of α emphasize more high utilization values.

In some cases, we may care only about imbalances that cause elements to be overloaded. Imbalances that do not cause overloading can be more easily tolerated, and may not require reconfiguration (especially if that reconfiguration is costly). We define a threshold function, $T(u_i^e,k)$, where k is a parameter defining the threshold utility value at which we consider an element to be overloaded.

Imbalance Function. Once we have compressed the temporal component of the load description, we use another function to calculate the imbalance factor. Instead of choosing a function directly, we first construct a set of requirements for such a function (listed in Table 2). These requirements formally capture the intuitive notion of *balanced*. Given these requirements, we develop a function, $f(\mathbf{L})$, that satisfies them. Note that there are multiple functions that satisfy the necessary criteria, but we consider only one here.

First, we normalize all load values $\lambda^e = L^e/\sum_{i=1}^n L^i$ and let $\mathbf{L} = \left\{\lambda^1,\lambda^2,\ldots,\lambda^n\right\}$ be the set of load values over all elements. None of the common statistical measures of dispersion satisfies all of the requirements, including range, variance, and standard deviation. One possible function that takes into account all of the properties in Table 2 is one based on entropy. We define the *normalized entropy* of a load distribution to be

$$f(\mathbf{L}) = 1 - \frac{\sum_{i=1}^n \left(\lambda^i \log \lambda^i\right)}{\log \frac{1}{n}}$$

where the numerator is the traditional definition of entropy, and the denominator is the normalizing factor. We orient the scale by taking the difference with 1.

Application: This module is independent of the target system. It acts only on the abstract *elements* and *load components* defined in Module 1.

2.3 Optimize Reconfiguration Plan

Once an imbalance has been detected, we determine a set of configuration changes that rebalance the load. The framework does not limit the kinds of configuration changes that are possible. Instead, the constraints are imposed by the system architecture. For example, this could include migrating a data unit such as single volume from one node to another or creating a load-balancing volume mirror. Each of these changes could potentially increase or decrease the load observed on each element in the system. We call a set of configuration changes a *reconfiguration plan*.

The goal of Module 3 is to determine a reconfiguration plan that minimizes both the imbalance of the system and the cost of the reconfiguration. Because of these two competing objectives, there may not be a single reconfiguration plan that optimizes both. Instead, we discover a range of reconfiguration plans, which emphasize each of the objectives to a different degree. Module 3 is further broken down into two independent components: evaluation, which calculates the objectives and total cost of any possible reconfiguration plan; and search, which determines which of the many possible reconfiguration plans to evaluate. In any practical scenario, it is not feasible to evaluate all possible reconfiguration plans, and so the search component must be more intelligent than exhaustive search.

There are many possible search techniques we could apply—one of our goals is to compare a number of these methods. We choose three methods: greedy algorithms, evolutionary (or genetic) algorithms, and integer programming. The following paragraphs outline how we estimate reconfiguration costs and describe the details of each of the three optimization methods applied within our framework.

Objectives and Costs. The objective of a system reconfiguration is to mitigate a load imbalance. Specifically, we seek to minimize the imbalance factor of the resulting load, while simultaneously minimizing the cost of the reconfiguration. We define the resulting load to be what the load would have been had the reconfigurations been made before the current load was observed. Calculating the cost of a reconfiguration is specific to the target system and is covered at the end of this section.

Greedy Algorithm. Greedy algorithms, such as hill climbing, are a search technique that combines a series of locally optimal choices to construct a complete solution. We are optimizing with respect to multiple objectives, and so there is usually not just a single optimal choice at each step. Our approach is to randomly select one of the non-dominated possibilities at each step. A non-dominated solution is one for which there is no other solution with both a lower cost and lower imbalance factor. Algorithm 1 defines our greedy approach. It is specific to data migration but can be easily adapted to other reconfiguration options.

Evolutionary Algorithm. Evolutionary algorithms work by maintaining a *population* of possible solutions, and explore the search space by recombining solutions that are found to perform better. In this way, they are similar to natural selection. We use an algorithm based on the Strength Pareto Evolutionary Algorithm (SPEA) [18], a type of multiobjective evolutionary algorithm (MOEA).

The algorithm uses three primary input parameters: the population size, s; the number of generations, n; and the archive size, m. The population size is the number of possible solutions considered at one time. One *generation* is defined as the process of creating a new population of solutions, by recombining solutions from the current population. So the total number of possible solutions evaluated by the algorithm is the product of the population size and number of generations. In general, increasing either

Algorithm 1. Greedy algorithm for load balancing by data migration.

```
let $s$, the system state currently under consideration, be the original system state.
initialize global solutions, $S = \{\text{current state}\}$.
repeat
    let $e_{max}$ and $e_{min}$ be the elements with maximum and minimum load (in $s$, respectively).
    initialize list of locally non-dominated solutions, $T = \emptyset$.
    for all load components $\ell$ on $e_{max}$ do
        let $t$ be the state of the system after migrating $\ell$ from $e_{max}$ to $e_{min}$.
        calculate the imbalance factor of $t$ and the cost of the migration.
        if $t$ is non-dominated with respect to $T$ and $S$, add $t$ to $T$.
    end for
    if $T \neq \emptyset$ then
        choose a random element $r \in T$.
        add $r$ to $S$.
        update the current state $s \leftarrow r$.
    end if
until $T = \emptyset$
```

of these two parameters will improve the quality of the final solutions, in exchange for a longer running time. The *archive* is a collection of the best (non-dominated) solutions seen by the algorithm at a given time. The size of the archive determines the final number of solutions produced by the algorithm.

Given the three input parameters m, n, and s, the algorithm generates as output archive A, a set of non-dominated solutions. At a high level, it works as follows:

1. **Initialize:** Create a population P of s possible reconfiguration plans, where each possible plan has a small number of random migrations specified.
2. **Evaluate:** Find the cost and imbalance factor for each solution in the current population P.
3. **Archive:** Add all non-dominated solutions from the population P to the archive A.
4. **Prune:** If the size of the archive A exceeds the maximum size m, remove some of the solutions based on measures of *crowding*. This is used to ensure the solutions take on the full range of possible values, in both imbalance factor and cost.
5. **Check Stopping Condition:** If the maximum number of generations has been reached, return A. Otherwise, continue.
6. **Select and Recombine:** Select individuals from the archive A and recombine them to form the next population P. A new solution is produced by combining two random subsets of migrations, each selected from an existing solution.
7. **Return** to Step 2.

Integer Programming. Another optimization technique used within our load-balancing framework is binary integer programming. Integer program solvers guarantee that the solution found will be optimal with respect to the given formulation. However, this method places several restrictions on such a formulation. The most important of these restrictions is that all of the equations, including the objective function, must be linear. That is, we cannot use arbitrary functions for the cost or imbalance functions.

Second, the method is not naturally multiobjective. To overcome this, we solve a series of integer-programming problems, with successively larger cost restrictions.

There are several variables and functions that describe our binary integer program. We use the following notation in its definition:

N_e The number of elements

N_l The number of load components

$$y_{ij} = \begin{cases} 0 & \text{if load component } i \text{ was originally assigned to element } j \\ 1 & \text{otherwise} \end{cases}$$

w_i the weight of load component i

c_{ij} the cost of migrating load component i to element j

C the maximum allowed cost of all migrations

U the target load on each element

Decision variables $x_1, \ldots, x_n$ are binary variables that the program will solve for.

$$x_{ij} = \begin{cases} 1 & \text{if load component } i \text{ assigned to element } j \\ 0 & \text{otherwise} \end{cases}$$

Objective function is a function of the form $\sum_{i=1}^{n} c_i x_i$, where c_i are any constants, and x_i are the decision variables. The goal is to maximize the total weight of all assigned load components:

$$\max \sum_{i=1}^{N_l} w_i \cdot \sum_{j=1}^{N_e} x_{ij}$$

Constraint functions are of the form $\sum_{i=1}^{m} c_i x_i < C$, where c_i are constants, x_i are some subset of the decision variables, and C is a constant.

Ensure that no load component is assigned to more than one element:

$$\forall i \leq N_l : \sum_{j=1}^{N_e} x_{ij} \leq 1$$

Ensure that maximum cost is not exceeded:

$$\sum_{i=1}^{N_l} \sum_{j=1}^{N_e} x_{ij} y_{ij} c_{ij} < C$$

Ensure that no element is overloaded:

$$\forall j \leq N_e : \sum_{i=1}^{N_l} x_{ij} w_{ij} \leq U$$

Note that it is possible that some load components will not be assigned to any element, either because doing so would exceed the target load or because it would exceed the maximum reconfiguration cost. These unassigned load components are assumed to remain on their original elements. The load components that the solver assigns to a new element make up the reconfiguration plan.

Application: We consider reconfiguration plans that consist of a set of volumes to be migrated from one Dblade to another (Scenario 2 from Section 2.1). The reconfiguration plans are evaluated with respect to two objectives: minimizing the imbalance factor calculated by Module 2; and minimizing the cost of the reconfiguration.

We define four functions for calculating cost, representing both linear and non-linear functions. The first function assigns a constant cost for each volume migration. The second function assigns a cost proportional to the total size of the volumes being moved. The third function uses empirically derived costs as encoded in a table-based model (see Table 3). The cost is the average latency of all operations while the migrations are taking place. This cost depends on both the number of volumes being simultaneously migrated and the workload being handled by the system at the time of reconfiguration. The fourth function is non-linear and estimates the total time of impairment.

Table 3 shows a sampling of (sanitized) values measured on a four-node cluster with midrange nodes running a SPECsfs benchmark [19]. The rows represent the relative

Table 3. Reconfiguration costs measured as average request latency in ms

Volume	*Load (ops/s)*				
Moves	Base (500)	2×	4×	8×	16×
0	0.5 ms	0.5	0.5	0.7	0.8
1	1.0 ms	0.8	0.9	1.0	1.1

Volume	*Load (ops/s)*				
Moves	Base (500)	2×	4×	8×	16×
2	1.2 ms	1.0	1.0	1.2	1.5
3	1.2 ms	1.2	1.3	1.5	1.9

costs when moving zero, one, two, or three volumes simultaneously. The columns represent the "load level" of the benchmark (in SPECsfs terms, the targeted number of operations per second). The baseline column corresponds to a very light load; the other columns represent a load-level that is a double, quadruple, and so on, of the baseline load. A performance engineering group generates similar tables for other cluster configurations, system versions and hardware types for other workload classes e.g., Exchange Server [20], during system development.

We implemented the hill-climbing and integer programming optimization methods directly in MATLAB. The Strength Pareto Evolutionary Algorithm (SPEA) is written primarily in Java$^{\text{TM}}$ and controlled by MATLAB. Hence, the SPEA runtime is dominated by interprocess communications. Because of this, we compare the efficiency of the methods by the number of possible solutions they evaluate, and not runtime.

The integer-programming method is restricted to using only linear cost and objective functions. It uses only a form of the simple sum function for calculating imbalance, and uses only the constant and linear cost functions. This corresponds to a form of traditional bin-packing problems. The greedy and evolutionary algorithms also correspond to traditional bin-packing when using the simple sum function and constant costs.

2.4 Execute Reconfiguration Plan

As described previously, Module 3 provides a set of (near) optimal reconfiguration plans, with a range of costs and resulting imbalance factors. The job of Module 4 is to select one of these possible solutions and perform a series of system-level operations to execute that plan. The choice of which of the nondominated solutions to choose depends primarily on the reconfiguration cost that the system can tolerate at the time, which can be specified by service level objectives (SLOs).

Application: In our case of performing load balancing through the migration of volumes, this module handles the details such as when to perform the migrations, in what order, and at what rate. In our experiments, we chose to perform all migrations simultaneously at the maximum rate. Our system performs this operation on-line with no client-perceived disruptions (though performance is affected, as described by Table 3).

2.5 Assumptions and Limitations

As presented here, the framework makes a few assumptions. First, we assume that the load on the system is sufficiently stable. That is, the load we have observed in the recent past is a good approximation of the load we will experience in the near future. If this were not the case, the steps taken to rebalance the system would likely be ineffective or even detrimental. If a system does not meet this criterion, the load-balancing framework would need some other means for predicting future load in order to determine a viable reconfiguration. However, in practice, enterprise system workloads tend to experience cyclical workloads with daily and weekly periodicities.

Second, we assume that the system is composed of (nearly) homogeneous nodes. For example, our model does not take into account the amount of memory each node has. We believe that our framework is applicable without any loss of generality to non-homogeneous systems. In practice, this requires more performance characterization with larger cost-model tables.

3 Experimental Analysis

We evaluate our framework in two different ways. First, using a description of a week-long workload from an internally deployed system, we explore the various implemented functions and optimization techniques. Second, we examine the model in a real-world scenario—a recent hardware upgrade and consolidation effort in one of our data centers.

We compare optimization methods for various imbalance and cost functions using a data center storage consolidation scenario. In this scenario, we study the effects of taking an existing stand-alone system with e-mail workload from MS Exchange Servers and integrating it into the existing six-node system. The first step involves rolling the existing hardware with its data into a cluster. The second step, and the one targeted by our framework, involves moving data (volumes) between the nodes of the combined system to achieve a more balanced load.

3.1 System and Workload Description

Clustered system. A six-node NetApp Data ONTAP GX system with 120 TB of storage across 794 disks and 396 total volumes stores the home directories of approximately 150 users, mostly engineers. Each user has a 200 GB primary volume with a remote site replica and a secondary RAID-protected volume with additional 200 GB of space that is not replicated. The home directory volumes are accessible by both NFS and CIFS protocols. The cluster is used predominantly for software development (compilation of source code) by individual users as well as build-server farms. There are several large volumes, one per cluster node, for a build- and regression-testing farm of a few dozen servers. We do not consider these volumes in our experiments because by manual assignment (and in practice), the load from these volumes is already "balanced." It is this kind of manual load balancing that we aim to replace with our framework.

Figure 2 shows the load on the cluster over a one-week period. The load is characterized by the total number of file operations performed by the system (aggregated over 30 minute periods). There were a total of approximately 1.4 billion operations, or an average of 8 million operations per hour. The load has a strong periodic component, with large spikes occurring during workdays, both during business hours (software development) and at night (regression tests). The bars at the top show the daily cumulative per-node load. Although the cluster has six nodes, it currently uses only the first four—the other ones have been recently added for future system expansion. We show later on how our framework redistributes load and populates these nodes with data.

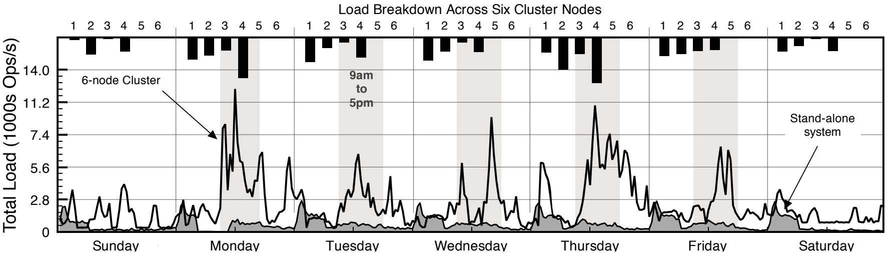


Fig. 2. Typical load profiles over a one-week period

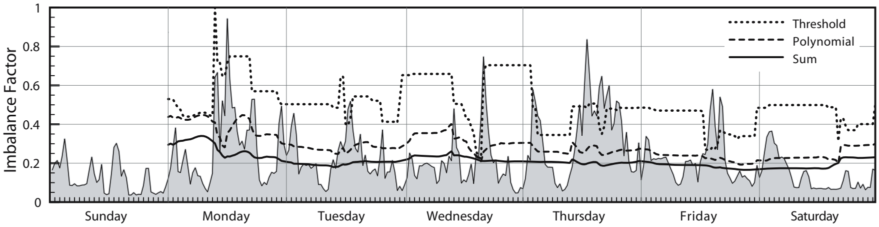


Fig. 3. Comparison of three load-flattening functions

Stand-alone system. There are 12 volumes supporting e-mail workload from three MS Exchange Servers with mailboxes for several hundred users. As shown in Figure 2, it also experiences load at night due to data consolidation and backup.

3.2 Results

Load Imbalance Expression. Figure 3 compares the three flattening functions from Module 2 with a 24-hour time window. Here, and for the remainder of this paper, we apply optimizations over a week-long period, since our workload has strong weekly periodicity. The input is the combined load from the two consolidated systems, shown in the background. The polynomial and threshold functions emphasize spikes in load more than the simple sum function, fulfilling their intended purpose. These two functions are also more variable in general, and require larger time windows to avoid rapid oscillations. Given our workload profile, a 24-hour window is sufficient to remove the daily load periodicity and more accurately reflect any underlying load imbalance.

Comparison of Optimization Methods. We use four cost functions—constant, linear, empirical, and non-linear; and three imbalance functions—sum, polynomial, and threshold, for our comparison of the different optimization methods in Figure 4. For the constant function, each volume move has a cost of 1. For the linear function, the cost of moving a set of volumes is the total number of bytes in all moved volumes. For the empirical function, we use the data from Table 3 and, more specifically, only the last column of the table with the base latency as 0.8 ms. Moving 1, 2, or 3 volumes increases the latency to 1.1, 1.5, or 1.9 ms, or respectively, 1.375, 1.875, and 2.375× the base.

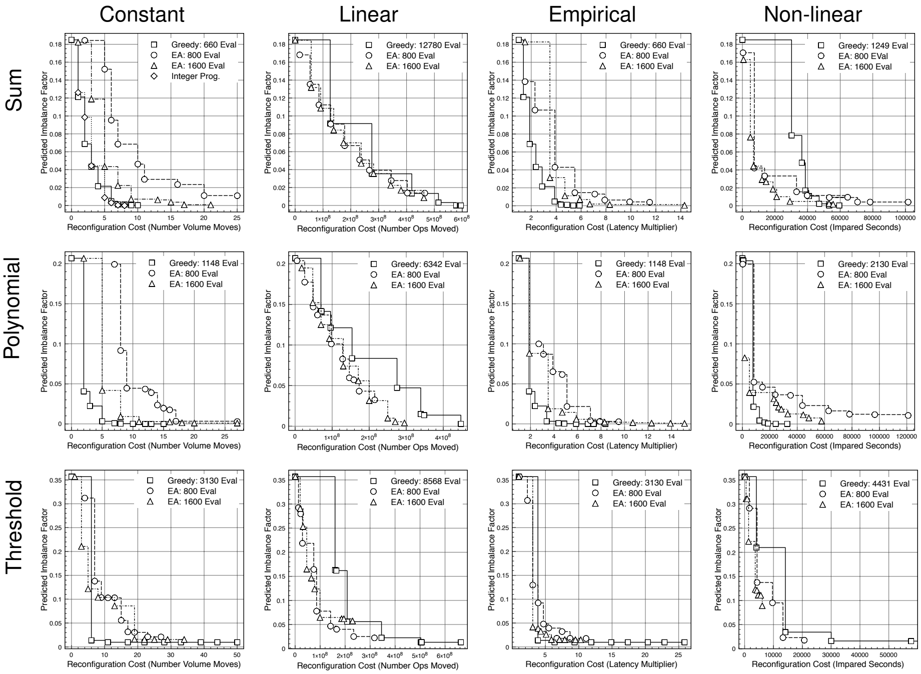


Fig. 4. Comparison of three optimization methods using various measures of imbalance and reconfiguration cost. The three rows correspond to the sum, polynomial, and threshold temporal-flattening functions. The four columns correspond to different cost models: constant (number of volumes moved); linear (number of bytes moved); empirical (latency increase); and, nonlinear (time period of impairment). The experiment using the sum-flattening function and constant migration cost compares all three optimization methods: greedy algorithm, evolutionary algorithm (EA), and integer programming. The others exclude integer programming because of their nonlinear objectives. We use two cases for EA with 800 and 1600 solution evaluations respectively. For all cases, solutions closer to the origin are better.

When moving more than three volumes, we interpolate using the last two values. So each volume move beyond three adds latency with a 0.5× latency multiplier. In practice, this linear penalty is too large for several volume moves. Therefore, we use data with approximate costs that are non-linear when moving more than three volumes simultaneously. The fourth column in Figure 4 shows the results for this approximation of a more realistic cost function.

The greedy algorithm uses an unbounded number of solution evaluations, halting when no further improvements can be made. The graphs display a set of 10 intermediate points along the hill-climbing path. The evolutionary algorithm uses a constant number of solution evaluations based on input parameter settings. The total number of evaluations is equal to the product of the population size and the number of generations. We show two settings: population size of 40 and 20 generations (for 800 evaluations) and population size 50 and 32 generations (for 1,600 evaluations). The results of the integer programming method are found in the experiment using only linear objectives (sum flattening function and constant cost).

We can draw the following general conclusions from the experimental results:

- The greedy algorithm is superior for the constant cost case. That is, it provides solutions with lower cost and lower imbalance factors.
- The evolutionary algorithm is superior for the linear cost case. With this more complex cost function, the greedy algorithm is more likely to be stuck in a local optimum early in the search, and requires more solution evaluations in total: in this case, up to 3,815 evaluations.
- The empirical cost function is nearly equivalent to the constant cost function. This is because each additional volume move adds a nearly constant latency multiplier, around 0.5×. Consequently, the experimental results of the third column are very similar to the first column. The greedy results are identical, with the same solution curves and number of evaluations. The evolutionary algorithm returns different solutions, due to the randomized starting point.
- The greedy algorithm tends to require more evaluations as the imbalance functions get more complex, moving from sum to polynomial to threshold.
- The integer-programming results are comparable with those found by the greedy algorithm. Some of its solutions perform slightly worse because the predicted imbalance of all solutions are evaluated using the normalized entropy function, but this nonlinear function is not used by the integer program.

Executing a Reconfiguration Plan. The optimization provides a set of nondominated solutions. The choice of which is most suitable depends on how much additional load (e.g., the expected increase in request latency) the system can tolerate during data migration. These are set as service-level objectives (SLOs) by a system administrator, allowing the reconfiguration plan to be executed automatically. If a single reconfiguration plan cannot reach a sufficiently balanced state without violating some SLOs, the framework iterates the process, as shown in Figure 1. By executing a small number of

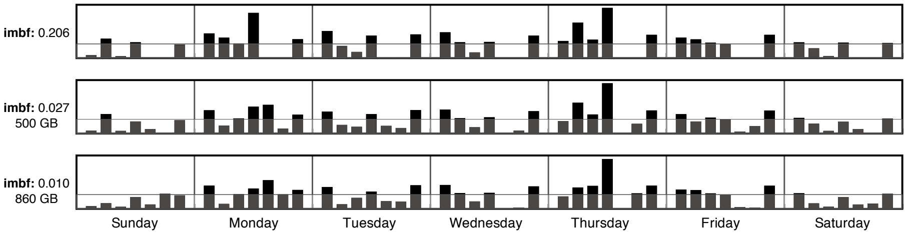


Fig. 5. Comparison of predicted load imbalance after reconfiguration and plan execution. The top graph is the baseline with imbalance factor (imbf) 0.207 for the combined load from both systems. The middle graph shows the reconfiguration plan with total cost of 500 GB. and the bottom graph shows the "best" reconfiguration plan—860 GB of data moved. A single volume contributes to Thursday's load on node 4; the spike cannot be flattened by data migration alone. Both graphs correspond to evolutionary algorithm solutions with the polynomial flattening function and linear cost function. The line at 1/3 graph height is a visual aid to more easily spot differences.

migrations over a longer time, or during off-peak hours, load imbalance can be eliminated without sacrificing system performance.

Figure 5 shows the effects of a subset of reconfiguration plans suggested by the evolutionary algorithm. These graphs illustrate how the load on each cluster node would change as a result of implementing a reconfiguration plan. The "best" plan at the bottom would be executed in about eight hours.

4 Related Work

Many components of our framework build upon previous work. Aqueduct, a tool for executing a series of data-migration tasks, with the goal of minimizing their effect on the current foreground work [21], is similar to Module 4 of our framework. The table lookup model in Module 3 is based on previous work of Anderson [22]. Our framework is similar in many aspects to Hippodrome [16], which iteratively searches over different system designs and uses Ergastulum to find the least-cost design in each step. Ergastulum is a system for designing storage systems with a given workload description and device model [13]. It uses a form of bin packing and solves the problem using a randomized greedy algorithm. Ergastulum was motivated by and improved upon Minerva [23], which uses a similar problem formulation but less-efficient search. Both of these systems focus on new system design for a specific workload. In contrast, our framework searches for load-balanced configurations of already-deployed systems, where reconfiguration cost if of importance. It is also suited for exploring what-if scenarios for system consolidation and upgrades. Stardust, which performs a function similar to Module 1, collects detailed per-request traces as requests flow through node and cluster components [11]. Other approaches mine data collected by Stardust for exploring what-if scenarios [10,24]. However, unlike our system, they use simple heuristics rather than optimization techniques. Our framework uses only high-level workload descriptions and performance levels similar to relative fitness models [25].

5 Conclusions

The modularity of our framework allows users to explore functions that best fit their workloads and systems. The use of multiple optimization methods *and* explicitly taking into account the cost of rebalancing when considering optimal configurations is one of the contributions of this work. Previous approaches have chosen a single method and designed their load-balancing systems around it.

To the best of our knowledge, we show the first application of evolutionary algorithms for optimizing storage system configurations. While not provably optimal, evolutionary algorithm is in our view the most general and versatile approach; it can leverage nonlinear imbalance functions and empirical system models. Integer programming is most applicable with simple, that is, linear, cost and objective functions and with fewer elements and load components.

Our framework is practical in terms of (i) using high-level workload descriptions from periodic collections of performance data, (ii) its applicability to real-world scenarios for consolidating data center storage, and (iii) the use of high-level empirical performance models. Generating detailed storage models is typically quite difficult. In contrast, collecting performance data for the table-based lookup model is "easy", though it can be resource intensive and time consuming. System characterization under different system configurations, workloads, and operations (e.g., volume moves) is an integral part of system development similar to qualifications of a storage system against a myriad of client host controller cards, operating systems, and so on. Dedicated engineering teams across manufacturers of enterprise storage routinely undergo such tests.

References

1. Eisler, M., Corbett, P., Kazar, M., Nydick, D., Wagner, C.: Data ONTAP GX: A scalable storage cluster. In: Proc. of the 5^{th} Conf. on File and Storage Technologies, pp. 139–152. USENIX Association (2007)
2. Abd-El-Malek, M., et al.: Ursa Minor: versatile cluster-based storage. In: Proc. of the 4^{th} Conf. on File and Storage Technologies, pp. 1–15. USENIX Association (2005)
3. Nagle, D., Serenyi, D., Matthews, A.: The Panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage. In: Proc. of the ACM/IEEE Conf. on Supercomputing, Washington, DC, USA, p. 53. IEEE Computer Society, Los Alamitos (2004)
4. Hitachi: Archivas: Single fixed-content repository for multiple applications (2007), http://www.archivas.com:8080/product_info/
5. Weil, S., Brandt, S., Miller, E., Long, D., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: Proc. of the 7^{th} Symposium on Operating Systems Design and Implementation, pp. 22–34. USENIX Association (2006)
6. Saito, Y., Frølund, S., Veitch, A., Merchant, A., Spence, S.: FAB: building distributed enterprise disk arrays from commodity components. In: Proc. of ASPLOS, pp. 48–58 (2004)
7. Litwin, W.: Linear Hashing: A new tool for file and table addressing. In: Proc. of the 6^{th} Int'l Conf. on Very Large Data Bases, pp. 212–223. IEEE Computer Society, Los Alamitos (1980)
8. IBM Corp.: TotalStorage productivity center with advanced provisioning (2007), http://www-03.ibm.com/systems/storage/software/center/provisioning/index.html
9. NetApp Inc.: NetApp storage suite: Operations manager (2007), http://www.netapp.com/products/enterprise-software/manageability-software/storage-suite/operations-manager.html
10. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for request extraction and workload modelling. In: OSDI, pp. 259–272 (2004)
11. Thereska, E., et al.: Stardust: tracking activity in a distributed storage system. SIGMETRICS Perform. Eval. Rev. 34(1), 3–14 (2006)
12. Thereska, E., et al.: Informed data distribution selection in a self-predicting storage system. In: Proc. of ICAC, Dublin, Ireland (2006)
13. Anderson, E., Spence, S., Swaminathan, R., Kallahalla, M., Wang, Q.: Quickly finding near-optimal storage designs. ACM Trans. Comput. Syst. 23(4), 337–374 (2005)
14. Weisstein, E.: Bin-packing and knapsack problem. MathWorld (2007), http://mathworld.wolfram.com/
15. Wilkes, J.: Traveling to Rome: QoS specifications for automated storage system management. In: Proc. of the Int'l. Workshop on Quality of Service, pp. 75–91. Springer, Heidelberg (2001)
16. Anderson, E., et al.: Hippodrome: Running circles around storage administration. In: Proc. of the 1^{st} Conf. on File and Storage Technologies, pp. 1–13. USENIX Association (2002)

17. Keeton, K., Kelly, T., Merchant, A., Santos, C., Wiener, J., Zhu, X., Beyer, D.: Don't settle for less than the best: use optimization to make decisions. In: Proc. of the 11^{th} Workshop on Hot Topics in Operating Systems. USENIX Association (2007)
18. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In: Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN 2001), International Center for Numerical Methods in Engineering (CIMNE), pp. 95–100 (2002)
19. SPEC: SPEC sfs benchmark (1993)
20. Garvey, B.: Exchange server 2007 performance characteristics using NetApp iSCSI storage systems. Technical Report TR-3565, NetApp., Inc. (2007)
21. Lu, C., Alvarez, G., Wilkes, J.: Aqueduct: Online data migration with performance guarantees. In: Proc. of the 1^{st} Conf. on File and Storage Technologies, p. 21. USENIX Association (2002)
22. Anderson, E.: Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-4 (2001)
23. Alvarez, G., et al.: Minerva: An automated resource provisioning tool for large-scale storage systems. ACM Transactions on Computer Systems 19(4), 483–518 (2001)
24. Thereska, E., Narayanan, D., Ganger, G.: Towards self-predicting systems: What if you could ask what-if? In: Proc. of the 3^{rd} Int'l. Workshop on Self-adaptive and Autonomic Computing Systems, Denmark (2005)
25. Mesnier, M., Wachs, M., Sambasivan, R., Zheng, A., Ganger, G.: Modeling the relative fitness of storage. In: Proc. of the Int'l. Conf. on Measurement and Modeling of Computer Systems. ACM Press, New York (2007)

Construction and Evaluation of Coordinated Performance Skeletons

Qiang Xu and Jaspal Subhlok*

University of Houston
Department of Computer Science
Houston, TX 77204
jaspal@uh.edu

Abstract. Performance prediction is particularly challenging for dynamic environments that cannot be modeled well due to reasons such as resource sharing and foreign system components. The approach to performance prediction taken in this work is based on the concept of a performance skeleton which is a short running program whose execution time in any scenario reflects the estimated execution time of the application it represents. The fundamental technical challenge addressed in this paper is the automatic construction of performance skeletons for parallel MPI programs. The steps in the skeleton construction procedure are 1) generation of process execution traces and conversion to a single coordinated logical program trace, 2) compression of the logical program trace, and 3) conversion to an executable parallel skeleton program. Results are presented to validate the construction methodology and prediction power of performance skeletons. The execution scenarios analyzed involve network sharing, different architectures and different MPI libraries. The emphasis is on identifying the strength and limitations of this approach to performance prediction.

1 Introduction

Traditional performance prediction and scheduling for distributed computing environments is based on modeling of application characteristics and execution environments. However, this approach is of limited value in some dynamic and unpredictable execution scenarios as modeling is impractical or impossible for a variety of reasons. Some example scenarios are execution with sharing of network or compute resources, execution with varying number of available processors, or execution with new system architectures or software libraries.

A new approach to performance prediction in such foreign environments is based on the concept of a *performance skeleton* which is defined to be a short running program whose execution time in any scenario reflects the estimated execution time of the application it represents. When the performance skeleton of an application is available, an estimate of the application execution time in a new environment is obtained by simply executing the performance skeleton and appropriately scaling the measured

* This material is based upon work supported by the National Science Foundation under Grant No. ACI- 0234328 and Grant No. CNS-0410797.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 73–86, 2008.

skeleton execution time. The main challenge in this approach is automatic construction of performance skeletons from applications. Earlier work in this project developed basic procedures for construction of communication and memory skeletons and explored their usage in distributed environments [1,2,3].

This paper introduces *scalable* construction of *coordinated* performance skeletons and evaluates their ability to predict performance in a variety of execution scenarios. The skeletons developed are “coordinated” implying that a single SPMD skeleton program is constructed instead of a family of process level skeletons. Improved compression procedures were developed that allow fast and nearly linear time skeleton construction. Validation experiments were conducted in a wide variety of scenarios including shared network bandwidth, shared processors, variable number of processors, different cluster architectures, and different MPI communication libraries. The results highlight the power and limitations of this approach.

We outline the procedure for the construction of performance skeletons for parallel MPI programs. Clearly a performance skeleton must capture the core execution and communication characteristics of an application. The skeleton construction procedure begins with the generation of process traces of an MPI application, primarily consisting of the message passing calls interspersed with computation segments. The first processing step is *trace logicalization* which is the conversion of the suite of MPI process level execution traces into a single logical trace. This is followed by *trace compression* which involves identification of the loop structure inherent in the execution trace to capture the core execution behavior. Final *skeleton construction* consists of generation of a deadlock free skeleton SPMD program from the compressed logical trace. The key steps are illustrated in Figure 1.

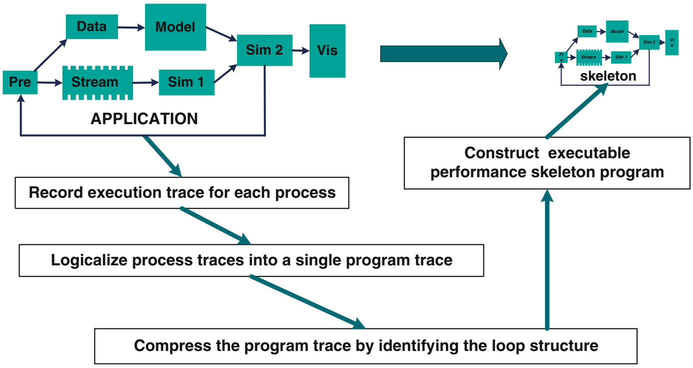

Data
Model
Sim 2
Vis
Pre
Stream
Sim 1
APPLICATION
skeleton
Record execution trace for each process
Logicalize process traces into a single program trace
Compress the program trace by identifying the loop structure
Construct executable performance skeleton program

Fig. 1. Skeleton construction

The state of the art in performance prediction and scheduling for distributed computing environments is based on modeling of application characteristics and execution environments, with some example systems discussed in [4,5,6]. The research presented in this paper is fundamentally different in being based on synthetically generated executable code as the primary vehicle for performance prediction. Trace analysis has also been addressed in the context of trace replay tools such as the work in [7].

DIMEMAS [8] presents a promising approach to performance prediction in distributed environments based on replay of an execution trace in a simulated environment. The underlying problems addressed in skeleton construction have many aspects in common with on-the-fly trace compression methods, in particular the work presented in [9,10]. However, the approaches are algorithmically different; specifically the approach presented performs logicalization first and compresses only the logical trace.

The paper is organized as follows. Section 2 presents the procedure for logicalization of MPI traces and section 3 presents the procedures developed for the compression of the logical trace. Section 4 introduces deadlock free skeleton program generation from the compressed trace. Section 5 presents and discusses results from the application of performance skeletons for performance prediction. Section 6 contains conclusions.

2 Trace Logicalization

As high performance scientific applications are generally SPMD programs, in most cases, the traces for different processes are similar to each other and the communication between processes is associated with a well defined global communication pattern. A study of DoD and DoE HPC codes at Los Alamos National Labs [11] and analysis of NAS benchmarks [12] shows that an overwhelming majority of these codes have a single low degree stencil as the dominant communication pattern. These characteristics expose the possibility of combining all processor traces into a single *logical program trace* that represents the aggregate execution of the program - in the same way as an SPMD program represents a family of processes that typically execute on different nodes. For illustration, consider the following sections of traces from a message exchange between 4 processes in a 1-dimensional ring topology.

Process 0	**Process 1**	**Process 2**	**Process 3**
...	...	...	...
snd(P1,...)	*snd(P2,...)*	*snd(P3,...)*	*snd(P0,...)*
rcv(P3,...)	*rcv(P0,...)*	*rcv(P1,...)*	*rcv(P2,...)*
...	...	...	...

The above physical trace can be summarized as the following logical trace:

Program
...
snd(P_R*,...)*
rcv(P_L*,...)*
...

where P_L and P_R refer to the logical left and logical right neighbors, respectively, for each process in a 1-dimensional ring topology.

Beside reducing the trace size by a factor equal to the number of processes, the logical program trace captures the parallel structure of the application. Note that this logicalization is orthogonal to *trace compression* discussed in the following section.

The logicalization framework has been developed for MPI programs and proceeds as follows. The application is linked with the PMPI library to record all message exchanges during execution. Summary information consisting of the number of messages and bytes exchanged between process pairs is recorded and converted to a binary *application communication matrix* that identifies process pairs with significant message traffic during execution. This matrix is then analyzed to determine the application level communication topology. Once this global topology is determined, a representative process trace is analyzed in detail and transformed into a logical program trace where all message sends and receives are to/from a logical neighbor in terms of a logical communication topology (e.g a torus or a grid) instead of a physical process rank. An example physical trace and the corresponding logical trace are shown in Table 1.

Table 1. Logical and physical trace for the 16-process BT benchmark

PHYSICAL TRACE	***LOGICAL TRACE***
....	....
MPI_Isend(... **1**, MPI_DOUBLE, 480, ...)	MPI_Isend(...**EAST**, MPI_DOUBLE, 480, ...)
MPI_Irecv(... **3**, MPI_DOUBLE, 480, ...)	MPI_Irecv(...**WEST**, MPI_DOUBLE, 480, ...)
MPI_Wait() /* wait for Isend */	MPI_Wait() /* wait for Isend */
MPI_Wait() /* wait for Irecv */	MPI_Wait() /* wait for Irecv */
....	....
MPI_Isend(... **4**, MPI_DOUBLE, 480, ...)	MPI_Isend(...**SOUTH**, MPI_DOUBLE, 480, ...)
MPI_Irecv(...**12**, MPI_DOUBLE, 480, ...)	MPI_Irecv(...**NORTH**, MPI_DOUBLE, 480, ...)
MPI_Wait() /* wait for Isend */	MPI_Wait() /* wait for Isend */
MPI_Wait() /* wait for Irecv */	MPI_Wait() /* wait for Irecv */
....	....
MPI_Isend(... **7**, MPI_DOUBLE, 480, ...)	MPI_Isend(...**SOUTHWEST**, MPI_DOUBLE, 480, ...)
MPI_Irecv(...**13**, MPI_DOUBLE, 480, ...)	MPI_Irecv(...**NORTHEAST**, MPI_DOUBLE, 480, ...)
MPI_Wait() /* wait for Isend */	MPI_Wait() /* wait for Isend */
MPI_Wait() /* wait for Irecv */	MPI_Wait() /* wait for Irecv */
....	....

The key algorithmic challenge in this work is the identification of the application communication topology from the application communication matrix which represents the inter-process communication graph. The communication topology is easy to identify if the processes are assigned numbers (or ranks) in a well defined order, but is a much harder problem in general. This is illustrated with a very simple example in Figure 2. The figure shows 9 executing processes with a 2D grid communication topology. In Figure 2(a) the processes are assigned numbers in row major order in terms of the underlying 2D grid. However, if the processes were numbered diagonally with respect to the underlying 2D grid pattern as indicated in Figure 2(b), the communication graph with process nodes laid out in row major order would appear as Figure 2(c). Clearly, the underlying 2D grid topology is easy to identify in the scenario represented in Figure 2(a) by a pattern matching approach but much harder when process numbering follows an unknown or arbitrary order, a relatively simple instance of which is the scenario represented in Figure 2(c). The state of the art in identifying communication topologies assumes that a simple known numbering scheme is followed [11].

The reasons topology identification is difficult are 1) establishing if a given communication graph matches a given topology is equivalent to solving the well known *graph*

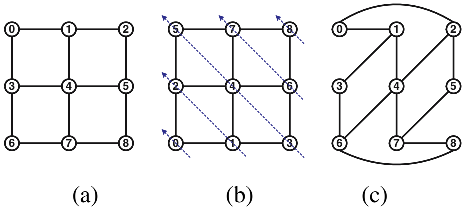


Fig. 2. 2D grid topology with row major and other numberings

isomorphism problem for which no polynomial algorithms exist and 2) there are many different types of topologies (different stencils on graph/torus, trees, etc.) and many instantiations within each topology type (e.g., different number and sizes of dimensions even for a fixed number of nodes). A framework consisting of the following tests was employed for topology identification:

1. **Simple tests:** All possible sizes of grid/tori/tree based on the number of processes N are identified with prime factoring. The number of edges and the degree ordered sequence of nodes for the given communication matrix are compared to instances of known topologies and those that do not match are eliminated.
2. **Graph Spectrum test:** Eigenvalue sets of isomorphic graphs are identical. The topologies whose eigenvalues do not match those of the communication matrix are eliminated.
3. **Graph Isomorphism test:** Graph isomorphism checking procedure is applied to establish that a given communication matrix exactly represents a specific topology. The *VF2* graph matching algorithm [13] was used.

The above steps are listed in increasing order of computation complexity and applied in that order as a decision tree. The simple tests and the graph spectrum test are employed to eliminate topologies that are provably not a match for the given communication matrix, but they cannot prove a match. Only the graph isomorphism test can establish an exact match.

Table 2 presents observations from the application of the logicalization procedure to selected NAS benchmarks running with 121/128 processes. The topologies that remain

Table 2. Identification of communication topologies of NAS benchmarks. Unique topologies are listed in boldface with other isomorphic topologies below them.

Benchmark (Processes)	Simple Tests	Graph Spectrum Test	Isomorphism Test	Trace Length Records(size)	Time (secs)
BT (121)	**11×11 6-p stencil**	**11×11 6-p stencil**	**11×11 6-p stencil**	50874 (2106KB)	30.76
SP (121)	**11×11 6-p stencil**	**11×11 6-p stencil**	**11×11 6-p stencil**	77414 (3365KB)	49.16
LU (128)	**16×8 grid**	**16×8 grid**	**16×8 grid**	203048 (9433KB)	134.30
CG (128)	**3-p stencil** **16×2×2×2 grid**	**3-p stencil**	**3-p stencil**	77978 (3224KB)	47.89
MG (128)	**8×2×2×2×2 torus** 8×4×2×2 torus 8×4×4 torus	**8×2×2×2×2 torus** 8×4×2×2 torus 8×4×4 torus	**8×2×2×2×2 torus** 8×4×2×2 torus 8×4×4 torus	9035 (386KB)	7.33

as candidates after each test and the final established topology are listed in columns corresponding to the tests. The trace length and size as well as the trace processing times are also listed. The tracing overhead is low as only gross communication data, such as the number of messages and bytes exchanged are recorded and analyzed. It is clear that the simple tests are very effective in reducing the number of topologies that are possible match candidates which is the key to the overall efficiency of the framework. The logicalization process and performance characteristics are described in detail in [14].

3 Trace Compression

An important step in the process of construction of performance skeletons is the identification of repeating patterns in MPI message communication. Since the MPI communication trace is typically a result of loop execution, discovering the executing loop nest from the trace is central to the task of skeleton construction. The discovery of "loops" here technically refers to the discovery of tandem repeating patterns in a trace (presumably) due to loop execution.

Common compression procedures include *gzip* [15] that constructs a dictionary of frequently occurring substrings and replaces each occurrence with a representative symbol, and *Sequitur* [16,17] that infers the hierarchical structure in a string by automatically constructing and applying grammar rules for reduction of substrings. Such methods cannot always identify long range loop patterns because of early reductions. An alternate approach is to attempt to identify the longest matching substring first. However, simple algorithms to achieve this are at least quadratic in trace length and hence impractical for long traces. A practical tradeoff is to limit the window size for substring matching, which again risks missing long span loops [9].

Our research took a novel approach to identifying the loop structure in a trace based on Crochemore's algorithm [18] that is widely used in pattern analysis in bioinformatics. This algorithm can identify all repeats in a string, including tandem, split, and overlapping repeats, in $O(nlogn)$ time. A framework was developed in this research to discover the loop nest structure by recursively identifying the longest span tandem repeats in a trace. The procedure identifies the optimal (or most compact) loop nest in terms of the span of the trace covered by loop nests and the size of the compressed loop nest representation. However, the execution time was unacceptable for long traces; processing of a trace consisting of approximately 320K MPI calls took over 31 hours.

The results motivated us to develop a greedy procedure which intuitively works bottom up - it selectively identifies and reduces the shorter span inner loops and replaces them with a single symbol, before discovering the longer span outer loops. While the loop nest discovered by the greedy algorithm may not be optimal, it has well defined theoretical properties. A key analytical result is that the reduction of a shorter span inner loop as prescribed in the greedy algorithm can impact the discovery of a longer span outer loop only in the following way: if the optimal outer loop is L_o then a corresponding loop L_g will be identified despite the reduction of an inner loop. L_o and L_g have identical but possibly reordered trace symbols, but L_g may have up to 2 less loop iterations than L_o. Hence, the loop structure discovered by the greedy algorithm is *near optimal*. The theoretical basis for this procedure is treated in depth in [19].

Table 3. Results for optimal and greedy compression procedures

Name	Raw Trace Length	Compression Time		Major Loop Structure	Trace Span Covered by Loops	Compressed Trace Length	Compression Ratio
		Greedy (secs)	Optimal (secs)				
BT B/C	17106	8.91	311.18	$(85)^{200} = (13 + (4)^3 + ... + (4)^3)^{200}$	99.38%	44	388.77
SP B/C	26888	7.61	747.73	67^{400}	99.67%	89	302.11
*CG B/C	41954	8.48	2021.78	$(552)^{75} = ((21)^{26} + 6)^{75}$	98.68%	10	4195.4
MG B	8909	8.64	113.48	$(416)^{20}$	93.39%	590	15.1
MG C	10047	10.88	144.54	$(470)^{20}$	93.56%	648	15.5
LU B	203048	33.16	44204.82	$(812)^{249} = ((4)^{100} + (4)^{100} + 12)^{249}$	99.58%	63	3222.98
LU C	323048	61.9	113890.21	$(1292)^{249} = ((4)^{160} + (4)^{160} + 12)^{249}$	99.58%	63	5127.75

The optimal and greedy loop nest discovery procedures were implemented and employed to discover the loop nests in the MPI traces of NAS benchmarks. The key results are listed in Table 3. The loop nest structure is represented in terms of the number of loop elements and the number of loop iterations. As illustration, the CG benchmark loop structure is denoted by $(552)^{75} = ((21)^{26} + 6)^{75}$ implying that there is an outer loop with 75 iterations enclosing 552 elements in the form of an inner loop with 21 elements iterated 26 times, and another 6 elements. As expected, the optimal algorithm discovered perfect loop nests as validated by direct observation. The loop nests discovered by the greedy algorithm were, in fact, identical to the optimal loop nests except for a minor difference in the case of CG benchmark - the compressed trace had 21 symbols instead of 10 and the loop structure was slightly different. However, the time for greedy loop discovery was dramatically lower, down from 31 hours to 61 seconds for one trace. To the best of our knowledge, this is the first effort toward extracting complete loop nests from execution traces.

4 Construction of Performance Skeletons

The final step in building a performance skeleton is converting a logicalized and compressed trace into an executable program that recreates the behavior represented in the trace. The trace at this stage consists of a loop nest with loop elements consisting of a series of symbols, each symbol representing an MPI Call or computation of a certain duration of time. The trace is converted to executable C code with the following basic steps:

- The loop nest in the trace is converted to a program loop nest with the number of iterations reduced to match the desired skeleton execution time.
- The collective and point-to-point communication calls in the trace are converted to MPI communication calls that operate on synthetic data. The point to point calls generate a global stencil communication pattern matching the application topology.
- The computation sections are replaced by synthetic computation code of equal duration without regard to the actual computation characteristics.

The procedure is simplistic in reproducing computation. The instruction mix may be different and memory behavior is not reproduced. This is a limitation of the current work although memory skeletons have been investigated separately in [1].

A direct conversion of MPI trace symbols to MPI calls can result in executable code that may deadlock. The key issues in ensuring deadlock free communication in a skeleton program are as follows:

1. **Identifying *local* communication.** Most MPI calls in a logical trace are matched: there is a *Recv* in the trace corresponding to every *Send*. We refer to these calls as *global* and their inclusion in the performance skeleton leads to a stencil communication pattern across executing nodes. However, typically some unmatched MPI Send/Recv calls exist in a trace even when there is a dominant global communication pattern, i.e. there may be *Send to WEST* in the trace but no corresponding *Send to EAST*. Such calls are labeled *local* and ignored for code generation. An alternate approach is to match the local calls with synthetically generated calls. While local calls imply inaccuracy, they are rare in structured codes and necessary to ensure deadlock free execution. The procedure for marking communication calls as local or global is outlined in Figure 3. It is based on the basic deadlock free patterns of point to point communication which are 1) a non blocking Send/Recv with a matching Recv/Send before a corresponding Wait and 2) One or more blocking Send/Recv calls followed by matching Recv/Send calls. Note that in the latter case, the code generated for end nodes in the stencil is different from others, e.g. Send followed by Recv, when it is Recv followed by Send for all other nodes.
2. **Unbalanced global communication.** Even when a pair of communication calls is matched, it may not be balanced, meaning an MPI Send/Receive and its

```
  while next-call= First unmarked Send or Recv call in the code exists do
    if next-call is a non-blocking iSend (iRecv) then
      Let match-wait be the corresponding matching Wait call.
      Let match-call be the next matching Recv/iRecv (Send/iSend) in the code.
      if match-call is after match-wait or match-wait or match-call does not exist then
        Mark next-call as local communication.
      else
        Mark next-call and match-call as global communication.
      end if
    else
next-call is a blocking Send (Recv).
      Let match-call be the next matching Recv/Irecv (Send/Isend) in code.
      if  no match-call exists or there is a blocking Send or Recv between next-call and match-call then
        Mark next-call as local communication.
      else
        Mark next-call and match-call as global communication.
      end if
    end if
  end while
```

Note: Matching calls have the same datatypes and match in terms of the directions in a communication pattern, e.g, logical East and West in a 2D torus.

Fig. 3. Identification of Global and Local Send and Recv communication calls

corresponding MPI Receive/Send may not be equal in size. Analysis is employed to identify these and force a match by using the median message size of a Send and Recv.

5 Experiments and Results

A framework for automatic construction of performance skeletons has been implemented. Automatically generated skeletons were employed to estimate the execution time of corresponding applications in a variety of scenarios. Prediction accuracy was measured by comparing the predicted performance with actual application performance.

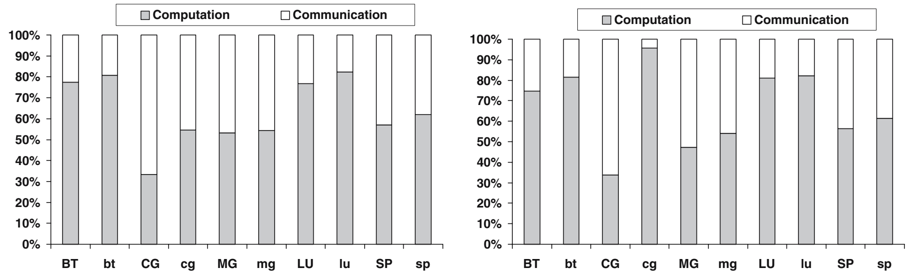
5.1 Skeleton Construction and Properties

Skeletons were constructed on "PGH201", a compute cluster composed of 10 Intel Xeon dual CPU 1.7 GHz machines with 100 Mbps network interfaces. The execution was under MPICH 2.0 library. Results are reported for 16-process class C NAS benchmarks. Execution is on 16 dedicated processors, except when noted otherwise. The methodology employed allows skeletons to be constructed to approximate a target skeleton execution time (or equivalently, a target ratio between application and skeleton execution times). However, there is a minimum execution time for a "good skeleton" which corresponds to the execution of a single iteration of the main execution loop. For the experiments conducted, the objective was to build the longest running skeleton with execution time under one minute or a skeleton that executes for approximately 10% of the application execution time, whichever was lower. The reference execution times of NAS benchmarks and their skeletons are shown in Table 4.

Table 4. Benchmark and skeleton execution times for NAS benchmarks on 16 processors

Benchmark	Execution Time(s)	
Name	Skeleton	Benchmark
BT	45.6	1129.6
CG	40.3	607.6
MG	8.3	79.1
LU	39.1	637.4
SP	43.1	1069.2

An application and the corresponding performance skeleton should have approximately the same percentage of time spent in computation and communication. These were measured for execution under MPICH 2.0 as well as execution under Open MPI library. The results are presented in Figure 4. We note that the computation/communication time percentage is generally very close for benchmarks and corresponding skeletons. One exception is the CG benchmark, where the difference is especially striking for execution under Open MPI. We will present the performance results for other benchmarks first and then specifically analyze the CG benchmark.

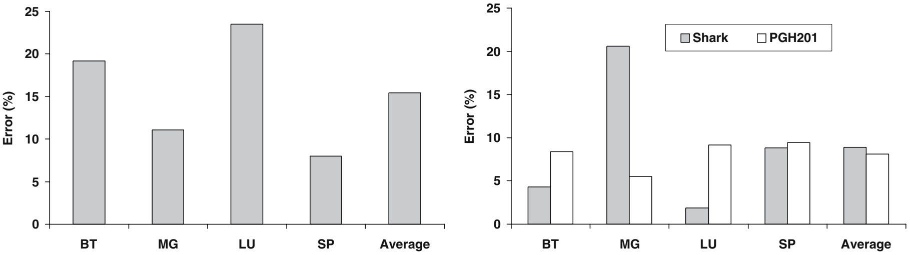


(a) MPICH 2.0 (b) OpenMPI

Fig. 4. Computation/communication time percentage for benchmarks (uppercase) and skeletons (lowercase)

5.2 Prediction Across MPI Libraries and Cluster Architectures

Skeletons constructed with MPICH 2.0 on PGH201 cluster were employed to predict performance under Open MPI library and on a different cluster called "Shark" which is composed of 24 SUN X2100 nodes with 2.2 GHz dual core AMD Opteron processor and 2 GB main memory. All nodes are connected through 4x InfiniBand Network Interconnect and Gigabit Ethernet Network Interconnect. The results are plotted in Figure 5.



(a) Across architectures (PGH201 to Shark) (b) Across libraries (MPICH 2.0 to OpenMPI)

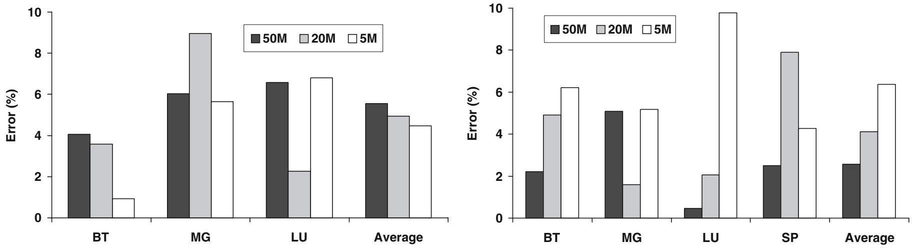
Fig. 5. Prediction results on 16 processors across MPI libraries/architectures

The prediction errors across the architectures average around 15%. The skeleton construction procedure employed makes no effort to reproduce the precise execution or memory behavior and only reproduces the execution times in skeletons with synthetic computation code. Hence, inaccuracy is expected across clusters with different processor and memory architectures. In the remainder of this paper, for validation purposes, the skeletons employed on Shark were "retuned" implying that the length of the computation blocks was adjusted to maintain the original ratio between reference skeleton and application execution.

Figure 5(b) shows the accuracy of performance predicted for OpenMPI with skeletons constructed with MPICH 2.0 on the two clusters. The errors are modest averaging below 10% for both clusters.

5.3 Prediction for Bandwidth Sharing

Figure 6 shows results from performance prediction with network sharing simulated by artificially reducing the available bandwidth to 50, 20, and 5Mbytes/sec with Linux *iproute2*. The results are presented for the older MPICH 1.2.6 MPI library, in addition to the MPICH 2.0 library. We consider the predictions to be excellent; the maximum prediction error is below 10% and the avarage prediction error varies between 2% and 6% for different scenarios. The results validate that the methodology employed models communication accurately.



(a) MPICH 1.2.6

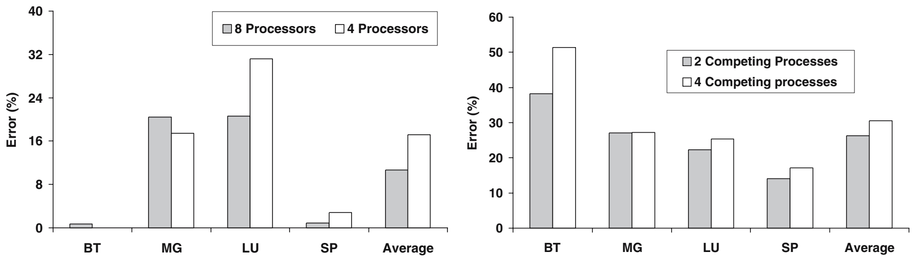
(b) MPICH 2.0

Fig. 6. Prediction results on 16 processors with reduced bandwidth availability

5.4 Prediction for Processor Sharing

A set of experiments was conducted to estimate the accuracy of performance prediction with processor sharing. Each node has an independent CPU scheduler and no gang scheduling is employed. First, 16 process jobs and corresponding skeletons were run on 8 and 4 processors. (The results are shown for the Shark cluster in this case as all cases cannot run on the PGH 201 cluster because of limited memory). The results in Figure 7(a) show that the average prediction error is around 10% for 8 processors and 5% for 4 processors, but the maximum errors are over 20% for 8 processors and over 30% for 4 processors. Figure 7(b) plots the accuracy of performance prediction on 16 processors with 2 or 4 synthetic competing compute bound processes on each node. The prediction errors are rather high averaging around 30%.

These results point out the limitation of the methodology employed as it does not model computation, synchronization, or memory behavior accurately. Performance with independent CPU schedulers and sharing is sensitive to these factors. We speculate that the main reason for the relatively low accuracy in the above scenarios is that the skeleton construction procedure does not model the idle periods caused due to synchronization accurately and some of them are replaced by computations in skeletons. In the case of processor sharing, the idle periods will be effectively used by other competing processes making the performance as predicted by skeletons to be inaccurate. In this set of experiments, errors were the result of the application execution times being less than those predicted by skeleton execution.



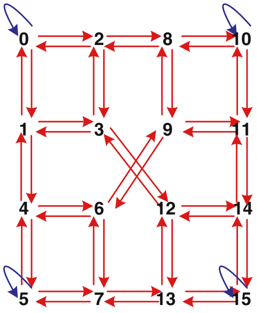
(a) Execution on 8/4 processors

(b) Execution with competing processes

Fig. 7. Prediction results for execution of 16 process jobs with processor sharing

5.5 CG Benchmark

The prediction errors for the CG benchmark were significantly higher than the rest of the benchmark suite for most scenarios, and the results were not included in earlier charts in order to streamline the discussion. As examples, the prediction error for CG was around 4 times the average for other benchmarks for prediction across libraries and prediction with reduced bandwidth. CG benchmark is very communication intensive and it was observed that the performance of the CG benchmark was very sensitive to the placement of processes on nodes. The communication topology of CG benchmark is shown on the left in Figure 8. The table on the right shows the execution time for various mappings of processes to nodes. The execution time varies by a factor of two depending on the location of the processes. The skeleton construction procedure makes no effort to manage placement of processes on nodes, and the placement for the skeleton can be different from the placement of the application. Since the performance is placement sensitive, the framework cannot deliver meaningful results. No other benchmark examined exhibited such strong sensitivity to process placement.



Config	Node 1	Node 2	Node 3	Node 4	Time(s)
A	0,1,8,9	2,3,10,11	4,5,12,13	6,7,14,15	496
B	0,1,4,5	2,3,6,7	8,9,12,13	10,11,14,15	568
C	0,1,2,3	4,5,6,7	8,9,10,11	12,13,14,15	272

Fig. 8. CG Topology and prediction results. The picture shows the communication topology. The table shows the execution time of the benchmark for various placements of processes on nodes.

6 Conclusions and Future Work

This paper has presented and evaluated a framework for the construction of performance skeletons for message passing MPI programs from execution traces. The objective is prediction of application performance in scenarios where modeling of performance is challenging. A key innovation is that the performance skeletons developed are *coordinated*, i.e., a single SPMD skeleton program is generated for a family of process level traces. The paper outlines the logicalization and compression procedures and lists related publications that contain the details.

Results presented validate the prediction ability of performance skeletons in different scenarios. It is observed that the skeletons are very effective in predicting performance when dynamics of communication change, e.g., when the bandwidth is limited or a new communication library is deployed. However, the prediction power is limited where the computation dynamics change, e.g., when multiple processes must share a processor. This is not entirely unexpected as the methodology captures the communication primitives precisely but attempts to recreate the periods of execution coarsely. If the computation regions in the skeleton were created to represent the instruction level execution and memory behavior, the approach would be significantly enhanced.

A basic limitation of this approach to performance prediction in its current form is that it is only applicable to structured applications with a repeating communication pattern for which a representative input data set is sufficient to capture the execution behavior. Extending this approach to unstructured applications and building skeletons that can simply take the data size as a parameter and predict performance appropriately are significant challenges to be addressed in future research.

References

1. Toomula, A., Subhlok, J.: Replicating memory behavior for performance prediction. In: Proceedings of LCR 2004: The 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems, Houston, TX, October 2004. The ACM Digital Library (2004)
2. Sodhi, S., Subhlok, J.: Automatic construction and evaluation of performance skeletons. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005), Denver, CO (April 2005)
3. Sodhi, S., Xu, Q., Subhlok, J.: Performance prediction with skeletons. Cluster Computing: The Journal of Networks, Software Tools and Applications 11(2) (June 2008)
4. Casanova, H., Obertelli, G., Berman, F., Wolski, R.: The AppLeS Parameter Sweep Template: User-level middleware for the grid. In: Supercomputing 2000, pp. 75–76 (2000)
5. Raman, R., Livny, M., Solomon, M.: Matchmaking: Distributed resource management for high throughput computing. In: 7th IEEE International Symposium on High Performance Distributed Computing (July 1998)
6. Snavely, A., Carrington, L., Wolter, N.: A framework for performance modeling and prediction. In: Proceedings of Supercomputing 2002 (2002)
7. Huband, S., McDonald, C.: A preliminary topological debugger for MPI programs. In: 1st International Symposium on Cluster Computing and the Grid (CCGRID 2001) (2001)
8. Badia, R., Labarta, J., Gimenez, J., Escale, F.: DIMEMAS: Predicting MPI applications behavior in Grid environments. In: Workshop on Grid Applications and Programming Tools (GGF8) (2003)

9. Noeth, M., Mueller, F., Schulz, M., de Supinskii, B.: Scalable compression and replay of communication traces in massively parallel environments. In: 21th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, CA (April 2007)
10. Ratn, P., Mueller, F., de Supinski, B., Schulz, M.: Preserving time in large-scale communication traces. In: 22nd ACM International Conference on Supercomputing, June 2008, pp. 46–55 (2008)
11. Kerbyson, D., Barker, K.: Automatic identification of application communication patterns via templates. In: 18th International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV (September 2005)
12. Tabe, T., Stout, Q.: The use of the MPI communication library in the NAS Parallel Benchmark. Technical Report CSE-TR-386-99, Department of Computer Science, University of Michigan (November 1999)
13. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: Performance evaluation of the VF graph matching algorithm. In: Proc. of the 10th ICIAP, vol. 2, pp. 1038–1041. IEEE Computer Society Press, Los Alamitos (1999)
14. Xu, Q., Prithivathi, R., Subhlok, J., Zheng, R.: Logicalization of MPI communication traces. Technical Report UH-CS-08-07, University of Houston (May 2008)
15. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)
16. Nevill-Manning, C., Witten, I., Maulsby, D.: Compression by induction of hierarchical grammars. In: Data Compression Conference, Snowbird, UT, pp. 244–253 (1994)
17. Nevill-Manning, C.G., Witten, I.H.: Sequitur, http://SEQUITUR.info
18. Crochemore, M.: An optimal algorithm for computing the repetitions in a word. Inf. Process. Lett. 12(5), 244–250 (1981)
19. Xu, Q., Subhlok, J.: Efficient discovery of loop nests in communication traces of parallel programs. Technical Report UH-CS-08-08, University of Houston (May 2008)

Data Sharing Analysis of Emerging Parallel Media Mining Workloads

Yu Chen[1], Wenlong Li[2], Junmin Lin[1], Aamer Jaleel[3], and Zhizhong Tang[1,*]

[1] Tsinghua National Laboratory for Information Science and Technology,
Department of Computer Science and Technology, Tsinghua University, Beijing, China
[2] Intel China Research Center, Beijing, China
[3] Intel Corporation, Hudson, MA
chenyu00@mails.tsinghua.edu.cn

Abstract. This paper characterizes the sharing behavior of emerging parallel media mining workloads for chip-multiprocessors. Media mining refers to techniques whereby users retrieve, organize, and manage media data. These applications are important in defining the design and performance decisions of future processors. We first show that the sharing behaviors of these workloads have a common pattern that the shared data footprint is small but the sharing activity is significant. Less than 15% of the cache space is shared, while 40% to 90% accesses are to the shared footprint in some workloads. Then, we show that for workloads with such significant sharing activity, a shared last-level cache is more attractive than private configurations. A shared 32MB last-level cache outperforms a private cache configuration by 20 – 60%. Finally, we show that in order to have good scalability on shared caches, thread-local storage should be minimized when building parallel media mining workloads.

1 Introduction

Processor and system architects use well chosen workloads to help design systems that perform well when running under targeted scenarios. As more hardware thread contexts are put on chip, future applications will exploit thread-level parallelism for higher performance and more functionality. Recognition, mining, and synthesis (RMS) workloads are emerging applications [2], where thread-level parallelism can be effectively exploited, thus are the design target of future multi-core processors. Within these workloads, one of the most important and growing application domain is the field of media mining, where workloads can extract meaningful knowledge from large amounts of multimedia data, to help end users search, browse, and manage enormous amounts of multimedia data [1].

As chip-multiprocessors (CMP) become pervasive and the number of cores increases, a key design issue will be the hierarchy and policies for the on-chip last-level

[*] This work was supported in part by National Natural Science Foundation of China under Grant No. 60573100,60773149.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 87–96, 2008.

cache (LLC). The most important application characteristics that drive this issue are the amount and type of sharing exhibited by the emerging workloads. This paper investigates the sharing behavior of the emerging parallel media mining applications. Our study makes the following contributions:

- Our study reveals that for most of the parallel media mining workloads, the sharing behaviors are in a common pattern that the shared data footprint is small but the sharing activity on the shared data is significant. The total portion of shared data is less than 15% while the portion of sharing accesses can be as large as 40 – 90% on some workloads. We then correlate this behavior with the nature of the algorithms in these workloads.
- Based on the sharing characteristics, we show that for these workloads, a shared LLC is preferred than private configurations on future CMPs. We investigate the performance of various configurations of LLC, and find that for most workloads, shared configurations can capture most amount of data-sharing and outperform private configurations by 20 – 60% in terms of cache misses.
- We show that minimizing the overhead of thread-local storage is important in parallelization and optimization to have good scalabilities on future CMPs with shared caches. For workloads which have large amount of sharing accesses with negligible thread-local storage, the performance of shared caches could scale well with increased thread count. On the contrary, for some workloads which involve large thread-local storage, scaling thread count from 4 to 16 can result in increase in cache miss rate from 0% to 18%, and decrease in sharing degree from 40% to 30%.

Both industry and academia have already invested resources in characterizing the scalability and performance of emerging data-mining applications. Chen et al. investigated the scalability and performance of emerging Tera-scale media mining applications [1]. Zambreno et al. composed the data-mining benchmark suite, Minebench, and analyzed some important performance characteristics on 8-way shared memory machines [12]. Jaleel et al. characterized the LLC performance of parallel bioinformatics workloads [4]. However, the sharing characteristics of media mining workloads revealed in this paper are very different from those of other domains such as bioinformatics.

2 Metrics and Methodology

We characterize the sharing behavior on the parallel media mining workloads by examine the following three metrics:

(i). Shared Cache Line: We define a *shared cache line* as one that is accessed during its lifetime by more than one thread.

(ii). Sharing Access: An access to a shared cache line is defined as a *sharing access*.

(iii). Type of Sharing: We further classify shared cache lines and sharing activities into *read-only* sharing and *read-write* sharing.

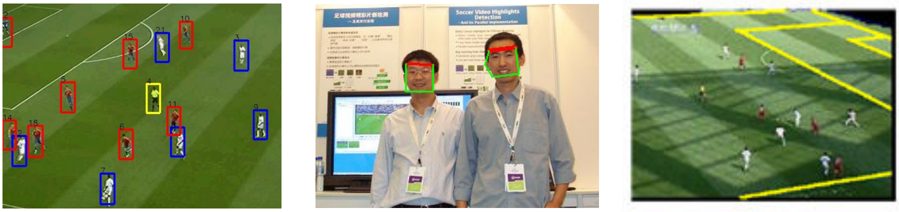
We use CMP$im, an instrumentation driven cache simulator for CMPs [3]. We focus on the sharing behavior of the parallel media mining workloads in this paper. The impact of latency on overall performance is part of our on-going work.

We assume a perfect instruction cache and unchanged private 32KB, 8-way L1 data caches. L2 cache (the last-level cache) is either private or shared, 8/16/32MB in size, and 16-way set associative. All caches use 64B line size, are non-inclusive, and use write-back and true LRU replacement policy. MSI invalidate-based coherence is maintained. We set the number of cores equals to the numbers of threads.

Our host machine is a 16-way Intel® Xeon® shared memory multi-processor (SMP) system. All workloads are compiled with optimization flags -O3 on a Windows 2003 32-bit system using Intel's C/C++ compilers.

3 Overview of the Workloads

The rapid advances in the hardware technology of media capture, storage, and computation power have contributed to an amazing growth in digital media content. As content generation and dissemination grows, extracting meaningful knowledge from large amounts of multimedia data becomes increasingly important. Media mining refers to a kind of technology whereby a user can retrieve, organize, and manage large amounts of multimedia data. It yields a wide range of emerging applications with various mass-market segments, e.g., image/video retrieval, video summarization, scene understanding, visual surveillance, etc. The six media mining workloads studied in this paper are from a real media mining system which is recently developed by Intel [1][5][6], and representative for the emerging media mining domain. Table 1 gives a description of workloads. Fig. 1 gives some examples of output frames.



Player Detection and Tracking Face Detection and Tracking Visual Keyword Detection

Fig. 1. Sample Output Frames from some Workloads

Table 1. Description of Workloads. IC is instruction count (in billion), and L1 MPKI is L1 misses per 1000 instructions.

Workload	Description	Algorithm	IC	L1 MPKI
Player Detection	Find multiple players within the playfield, track their moving trajectories, and identify their labels (two teams and referee) in the soccer video [8].	Players are first positioned by a person detector composed by boosted cascade Haar features [10]. Then based on unsupervised prior learned player appearance models, each player is automatically categorized.	158.4	17.25

Table 1. (*continued*)

Ball Detection	Locate the ball and track its moving trajectory in soccer video [9].	The algorithm includes image thresholding, connect-component analysis, Kalman filtering, the shortest path finding via Dijkstra algorithm, etc.	152.9	32.06
Face Detection	Face tracking [7] is to detect person's continuous faces from a video sequence.	A boosting detector slides by a window over the input image to detect whether the window contain a face or not.	228.9	5.94
Goalmouth Detection	Locate the two vertical poles and the horizontal bar. Used for virtual advertisement insertion and scene analysis [11].	Image filtering and Hough transform, region growing, pole pairing and height constraint	117.1	7.43
Shot Detection	Detects shot boundary in videos. To analyze the video content semantically, shot boundary detection is a prerequisite step [5].	Color histograms are used to introduce spatial information. The pixel intensity, mean grey value, and motion vector of each frame are calculated.	108.3	7.52
Visual Keyword Detection	View type indicates play status of the sports game and presents scene transition context of semantic events. Key frames are classified into different types: global, medium, close-up, and out of view.	The corresponding low-level processing includes playfield segmentation by the HSV dominant color of playfield and connect-component analysis. The dominant color of the playfield is adaptively trained by the accumulation of the HSV color histogram on a lot of frames [6].	160.2	9.79

All these media mining workloads are naturally data intensive, evidenced by the high L1 cache misses shown on the last column of the table. The high ratio of memory instructions, especially memory read instructions, can be explained as these workloads work through large amount of multimedia data in order to discover meaningful patterns.

4 Characterization Results

4.1 Cache Space Utilization

We investigate the cache space utilization by measuring how much portion of cache space is occupied by private and shared data respectively, as shown in Fig. 2. All cache lines are classified into *unused*, *private*, *read-only*, and *read-write shared*. A cache line of unused type denotes that it is not filled with any valid data during whole execution. We count the number of cache lines of each type, and present the averaged data over the periodic logs generated by our cache simulator.

The overall footprint of shared data is small. From Fig. 2, we can see the total portion of shared data, including read-only shared and read-write shared, is less than 15%, except for Face Detection on an 8MB LLC. This can be explained by the way of

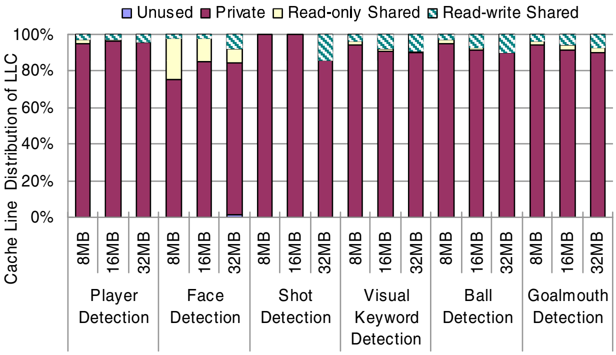


Fig. 2. Distribution of Last-level Cache Lines on Type of Sharing. The number of threads is set at 8, and the cache size varies from 8MB to 32MB. A cache line of unused type denotes that it is not filled with any valid data during whole execution.

processing video frames. The same arithmetic computation is applied to each frame independently in these workloads. For example, in Ball Detection, different threads work on different frames of the input video to perform video decoding and feature extraction. The threads only share a small global queue for synchronization purpose.

The shared data footprint is mainly consisting of read-write shared data, except for Face Detection. As we can see from the above figures, read-only shared data footprint is far smaller than read-write shared data footprint in Player Detection, Visual Keyword Detection, Ball Detection, and Goalmouth Detection. For Shot Detection, read-only shared data can hardly be seen. This is because in these workloads, the primary shared data structure is used for synchronization or task scheduling.

Based on Fig. 2, we can see that the amount of shared cache lines varies with the cache size. Increasing the cache size can either increase or decrease the portion of shared cache lines. For example, in Face Detection, shared data footprint goes down from 25% to 15% when the cache size varies from 8MB to 32MB. This indicates that the shared data footprint can be fit in to the cache and not be evicted by the private data in a smaller cache size. Second, increase in cache size may increase the portion of shared data footprints. This can be observed in Shot Detection, Visual Keyword Detection, Ball Detection, and Goalmouth Detection. For example, in Shot Detection, shared data footprint increases from 0% to 15% when the cache size varies from 16MB to 32MB. This behavior is due to the eviction of shared data by conflict and capacity miss of private data in smaller caches. We will further explain the eviction by comparing the access distribution on shared and private data in the following section.

4.2 Cache Access Distribution

We investigate the sharing activity by measuring how much portion of the last-level cache accesses is taken by sharing and private accesses respectively, as shown in Fig.3. All cache accesses are classified into *miss*, *private*, *read-only sharing*, and *read-write sharing*.

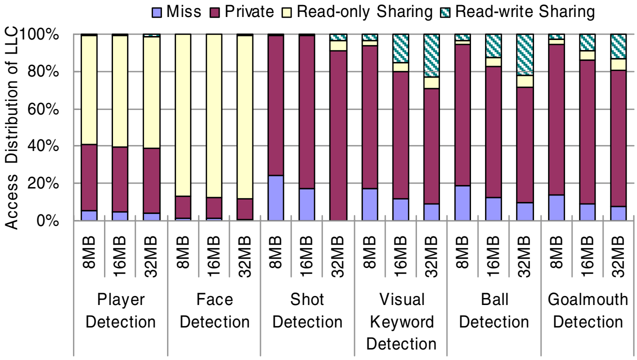


Fig. 3. Distribution of Last-level Cache Accesses on Type of Sharing. The number of threads is set at 8, and the cache size varies from 8MB to 32MB.

The overall amount of sharing activities is significant except for Shot Detection, but can be varying across different workloads. From Fig. 3, we can see that for Player Detection and Face Detection, the sharing accesses are as much as 40 – 90%, while for Visual Keyword Detection, Ball Detection, and Goalmouth Detection, the portion of sharing accesses varies from 5% to 30%. The higher portion of sharing accesses to private accesses in Player Detection and Face Detection indicates that the shared data is more frequently accessed than private data in these two workloads. This cross-validates the observation of shared data not being evicted by private data in a smaller cache discussed in the previous section. On the other hand, the lower portion of sharing accesses to private accesses in Visual Keyword Detection, Ball Detection, and Goalmouth Detection indicates that the shared data is less frequently accesses than private data. Thus shared data can be evicted by private data in smaller caches.

Further looking into the type of sharing, we can see that the major type of sharing varies from workloads to workloads. The sharing accesses mainly consist of read-only sharing in Player Detection and Face Detection, but mainly consist of read-write sharing in Visual Keyword Detection, Ball Detection, and Goalmouth Detection. The first two workloads need frequently read a global training model shared among their threads, while the last three workloads have to read from and write to a global synchronization or task scheduling data structure.

We can observe that the amount of sharing activity varies with the cache size in several workloads. A cache with more space can capture more sharing activity. For example, in Visual Keyword Detection, the portion of sharing access increases from 5% to 25% when the cache size increases from 8MB to 32MB. The same behaviors can also be observed in Shot Detection, Ball Detection and Goalmouth Detection. This is because shared data in these workloads is usually accessed in outer loops by operations such as reductions and synchronizations. Thus the time between two adjacent accesses to the same shared data element may become too large to let the data be reused in a smaller cache. For Player Detection and Face Detection, the amount of sharing activities remains roughly flat with varying cache size, indicating that the shared footprint in these two workloads can be fit into smaller caches.

Based on the cache space utilization and cache access distribution analysis in this and above sections, we conclude that these parallel media mining workloads have small shared data footprint but significant sharing activities on the shared data. The ratio of sharing accesses is much larger than the ratio of shared data footprint. We project that for these parallel media mining workloads, a shared last-level cache can deliver better performance than a private configuration, in terms of cache misses.

4.3 Performance Comparison between Private and Shared Caches

We compare the performance of shared and private cache configurations as shown in Fig. 4, the miss of private configuration with 8MB total size serves as the baseline, and all other configurations are normalized to this one. The performance of a private configuration is presented as the average performance of all private caches. From the figure, we can see that for Player Detection, Face Detection, Visual Keyword Detection, Ball Detection, and Goalmouth Detection, the shared cache configurations always outperform the private cache configurations by about 20 – 60%, in terms of cache misses. For Player Detection and Face Detection which exhibit large amount of read-only sharing, the higher miss rates of private caches come from the multiple misses of the same shared cache line. For Visual Keyword Detection, Ball Detection, and Goalmouth Detection which expose significant read-write sharing, the higher miss rates of private caches than shared caches come from the frequent invalidation of the read-write shared data. For Shot Detection, the shared cache configuration only provides marginal improvement, since it exhibits very small portion of both shared data footprint and sharing activities.

Based on the data presented, we conclude that for the parallel media mining workloads in our study, a shared LLC is preferred on future CMPs. Although most of the

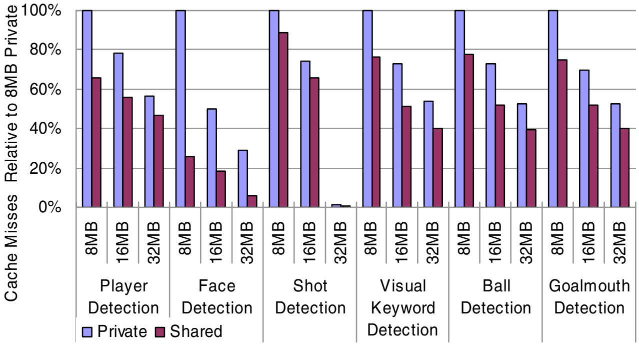


Fig. 4. Performance comparison between private and shared last-level cache configurations. The number of threads is set to 8, and the cache size is varying from 8MB to 32MB. A private cache configuration of S MB means there are totally 8 private caches, and the size of each cache is S/8 MB.

media mining workloads exhibit little shared data footprint, they have large amount of sharing activities, thus prefer a shared configuration of last-level cache. A shared last-level cache can utilize the inherent sharing behavior of these workloads, thus is more attractive for these workloads.

4.4 Effect of Scaling Thread Count

After demonstrating the benefit of shared LLCs, we now investigate the impact of varying the number of threads to the sharing behavior by comparing our metrics with varying thread counts, as shown in Fig. 5 and Fig. 6.

It can be observed that the amount of shared data footprint varies with the number of threads. As Fig. 5 illustrates, Shot Detection, Visual Keyword Detection, Ball Detection and Goalmouth Detection all experience significant decrease in sharing degree when adding more threads. This can be explained by the increasing amount of thread-local storage in these workloads. For example, in Ball Detection, each thread adds a private copy of large data structures storing intermediate computation result. As

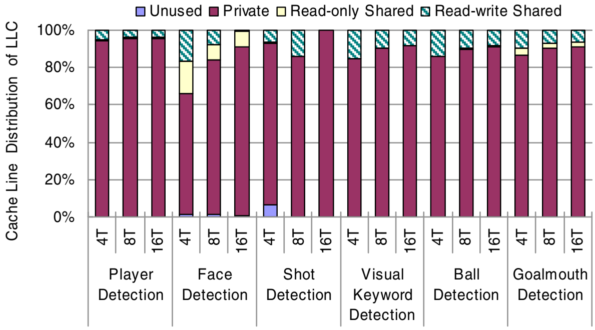


Fig. 5. Distribution of Last-level Cache Lines on Type of Sharing. The cache size is set at 32MB, and the number of threads varies from 4 to 16.

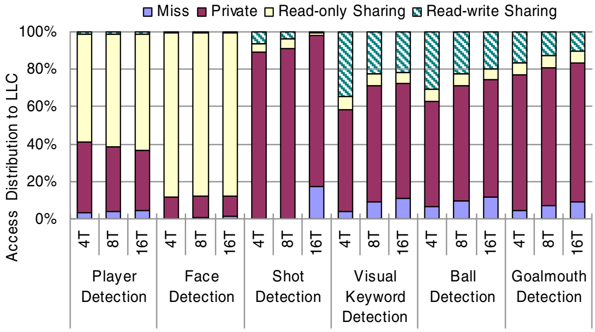


Fig. 6. Distribution of Last-level Cache Accesses on Type of Sharing. The cache size is set at 32MB, and the number of threads varies from 4 to 16.

the number of threads increase, the number of private copies also increases. Thus the portion of shared data footprint decreases with increasing thread count.

The increased footprint of private data could translate into the increase of cache misses and decrease in sharing activity, as shown in Fig. 6. For example, when scaling the number of threads from 8 to 16, the private data footprint of Shot Detection increased from 85% to almost 100%, and the cache misses in turn increase from 0% to 18%. In Visual Keyword Detection, the portion of sharing access decreases from 40% to 30% when the number of threads increases from 4 to 16. The same behavior can be observed in Shot Detection, Ball Detection, and Goalmouth Detection. This is because some of the shared data is evicted by the increased thread-local storage in these workloads. However, for Player Detection and Face Detection, the portion of sharing activities remains flat with increasing thread count, and these two workloads do not experience increase in cache miss.

Based on the observations from varying thread count, we conclude that minimizing the thread-local storage is important for building well scaling parallel media mining workloads on future CMPs with shared caches. For workloads which have significant amount of sharing activities, and have negligible thread-local storage, the cache performance of them can scale well when adding more thread contexts. On the other hand, for some workloads which involve large thread-local storage, scaling thread count can result in increase in cache miss rate and decrease in sharing degree.

5 Conclusion

The study in this paper gives insights into workloads that will be very important in future high-performance processors. We first show that most parallel media mining workloads studied in this paper exhibit a common sharing behavior that small shared data footprint but tremendous amount of data sharing activities. Thus, rather than partitioning the last-level cache into multiple private caches, a shared cache is more attractive in delivering better performance for media mining workloads on future high performance platforms. Furthermore, we find that in order to have good scalabilities on future CMPs with shared caches, the overhead of thread-local storages should be minimized when parallelizing and optimizing these workloads.

References

1. Chen, Y., Li, Q., Li, W., Wang, T., Li, J., Tong, X., Chen, Y., Zhang, Y., Hu, W., Wang, P.: Media Mining – Emerging Tera-scale Computing Applications. Intel Technology Journal (2007)
2. Dubey, P.: Recognition, Mining and Synthesis Moves Computers to the Era of Tera. Intel Technology Journal (February 2005)
3. Jaleel, A., Cohn, R., Luk, C., Jacob, B.: CMP$im: A Binary Instrumentation Approach to Modeling Memory Behavior of Workloads on CMPs. Technical Report-UMDSCA-2006-01 (2006)

4. Jaleel, A., Mattina, M., Jacob, B.: Last Level Cache (LLC) Performance of Data Mining Workloads On a CMP-A Case Study of Parallel Bioinformatics Workloads. In: 12th International Symposium on High Performance Computer Architecture (HPCA) (2006)
5. Li, E., Li, W., Wang, T., Di, N., Dulong, C., Zhang, Y.: Towards the Parallelization of Shot Detection - A Typical Video Mining Application Study. In: ICPP 2006, Columbus, Ohio, USA, August 14-18 (2006)
6. Li, W., Li, E., Dulong, C., Chen, Y.K., Wang, T., Zhang, Y.: Workload Characterization of a Parallel Video Mining Application on a 16-Way Shared-Memory Multiprocessor System. In: IEEE International Symposium on Workload Characterization (2006)
7. Li, Y., Ai, H., Huang, C., Lao, S.: Robust Head Tracking with Particles Based on Multiple Cues Fusion. In: Huang, T.S., Sebe, N., Lew, M., Pavlović, V., Kölsch, M., Galata, A., Kisačanin, B. (eds.) ECCV 2006 Workshop on HCI. LNCS, vol. 3979, pp. 29–39. Springer, Heidelberg (2006)
8. Liu, J., Tong, X., Li, W., Wang, T., Zhang, Y., Wang, H., Yang, B., Sun, L., Yang, S.: Automatic Player Detection, Labeling and Tracking in Broadcast Soccer Video. In: British Machine Vision Conference (2007)
9. Tong, X., Wang, T., Li, W., Zhang, Y., Yang, B., Wang, F., Sun, L., Yang, S.: A Three-Level Scheme for Real-Time Ball Tracking. In: Sebe, N., Liu, Y., Zhuang, Y.-t., Huang, T.S. (eds.) MCAM 2007. LNCS, vol. 4577, pp. 161–171. Springer, Heidelberg (2007)
10. Viola, P., Jones, M.: Rapid Object Detection using a Boosted Cascade of Simple Features. In: IEEE International Conference on Computer Vision and Pattern Recognition (CVPR) (2001)
11. Wan, K., Yan, X., Yu, X., Xu, C.: Real-Time Goal-Mouth Detection in MPEG Soccer Video. In: ACM Multimedia, pp. 311–314 (2003)
12. Zambreno, J., Ozisikyilmaz, B., Pisharath, J., Memik, G., Choudhary, A.: Performance characterization of data mining applications using MineBench. In: 9th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW) (2006)

Efficient PDM Sorting Algorithms[*]

Vamsi Kundeti and Sanguthevar Rajasekaran

Department of Computer Science and Engineering
University of Connecticut, Storrs, CT 06269, USA
{vamsik,rajasek}@engr.uconn.edu

Abstract. In this paper we present efficient algorithms for sorting on the *The Parallel Disks Model (PDM)*. Numerous asymptotically optimal algorithms have been proposed in the literature. However many of them have large underlying constants in the time bounds. We present practical and optimal algorithms in this paper. We have implemented these algorithms and evaluated their performance. Experimental data are very promising.

1 Introduction

Sorting is a fundamental problem that has numerous applications and hence has been studied extensively. When the amount of data to be processed is large, secondary storage devices such as disks have to be employed in which case the I/O becomes a bottleneck. To alleviate this bottleneck computing models with multiple disks have been proposed. One such model is the Parallel Disks Model (PDM). In a PDM, there is a (sequential or parallel) computer that has access to $D(\geq 1)$ disks. In one I/O operation, it is assumed that a block of size B can be fetched into the main memory from each disk. One typically assumes that the main memory has size M where M is a (small) constant multiple of DB.

A number of algorithms can be found in the literature for sorting on the PDM. Many of these algorithms are asymptotically optimal. However the underlying constants in the run time of many of these algorithms are high and hence it is not clear how practical these will be. In this paper we present algorithms that are asymptotically optimal and promising to be practical. We support our claim with experimental data.

The performance of any PDM sorting algorithm is measured in terms of the number of parallel I/O operations performed by the algorithm. The time taken for local computations is typically ignored since it tends to be much less than the time taken by I/O. A lower bound of $\Omega\left(\frac{N}{DB}\frac{\log(N/B)}{\log(M/B)}\right)$ on the number of parallel I/O operations to sort N keys has been proven by Aggarwal and Vitter [2].[1]

Notation. We say the amount of resource (like time, space, etc.) used by a randomized algorithm is $\widetilde{O}(f(N))$ if the amount of resource used is no more

[*] This research has been supported in part by the NSF Grant ITR-0326155 and a UTC endowment.

[1] In this paper we use log to denote logarithms to the base 2 and ln to denote logarithms to the base e.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 97–107, 2008.

than $c\alpha f(N)$ with probability $\geq (1 - N^{-\alpha})$ for any $N \geq n_0$, where c and n_0 are constants and α is a constant ≥ 1. We could also define the asymptotic functions $\widetilde{\Theta}(.)$, $\widetilde{o}(.)$, etc. in a similar manner.

2 Prior Algorithms and Our Results

Two kinds of algorithms can be found in the literature for PDM sorting. The first kind of algorithms (see e.g., [8,15,16]) distribute keys based on their values and hence are similar to bucket sort. The second kind of algorithms (see e.g., [1,4,5,9,10]) use R-way merging for a suitable value of R.

Ensuring full write parallelism is a challenge in distribution based algorithms and ensuring read parallelism is difficult in the case of merge based algorithms. The algorithm of Barve, Grove, and Vitter [4] ia based on merging and it uses a value of $R = M/B$. This algorithm called *Simple Randomized Mergesort (SRM)* stripes the runs across the disks. For each run the first block is stored in a random disk and the other blocks are stored in a cyclic fashion starting from the random disk. SRM has been shown to have an optimal expected performance only when the internal memory size M is $\Omega(BD \log D)$. No high probability bounds have been proven for SRM. The typical assumption made on M is that $M = O(DB)$.

SRM algorithm has been modified by Hutchinson, Sanders and Vitter [7]. Their approach is based on an algorithm of Sanders, Egner and Korst [14] who use lazy writing at the expense of an internal buffer. They employ Fully Randomized (FR) scheduling to allocate blocks of each stream to disks.An *expected* parallelism of $\Omega(D)$ is proven using asymptotic queuing theoretic analysis. Vitter and Hutchinson extended the above scheme to a scheduling scheme called Random Cycling (RC). The FR schedule is more complicated to implement and is not read-optimal for $M = o(BD \log D)$.

Many asymptotically optimal algorithms are known for sorting on the PDM (see e.g., [3], [9], [10], [15], etc.). Recently, Rajasekaran and Sen [11] have come up with practical and optimal algorithms for sorting on the PDM. In particular, they present an asymptotically optimal randomized algorithm for sorting. for the first time they have been able to obtain high probability bounds using basic principles.

In the next section we summarize the ideas in the algorithms of [11]. In this paper we present variants of the randomized algorithm presented in [11]. Our analysis indicates that the underlying constant in the run time of the new algorithm is less than that of [11]'s algorithm by 40%. Also, our implementation results show that the number of parallel I/Os made by the new algorithm is around 2.5 times the lower bound known on this number.

3 A Summary of Rajasekaran and Sen's Algorithm

The basic scheme of Rajasekaran and Sen's algorithm is to first randomly permute the given input sequence and then use a D-way merge on the permuted sequence. The random permutation is achieved using an integer sorting algorithm.

3.1 Integer Sorting

Consider the case when the keys to be sorted are integers in the range $[1, Q]$ for some integer Q. Integer sorting is a well studied problem in sequence and in parallel. For example, we can sort n integers in $O(n)$ time if the keys are in the range $[1, n^c]$ (for any constant c) (see e.g., [6]). There are two types of integer sorting algorithms, namely, forward radix sorting (or Most Significant Bit (MSB) first sorting) and backward radix sorting (or Least Significant Bit (LSB) first sorting). Rajasekaran and Sen [11] make use of backward radix sorting. The keys are sorted in phases where in each phase the keys are sorted only with respect to some number of bits.

The following Theorems are due to Rajasekaran and Sen [11]:

Theorem 1. *N random integers in the range $[1, R]$ (for any R) can be sorted in an expected $(1+\nu)\frac{\log(N/M)}{\log(M/B)}+1$ passes through the data, where ν is a constant < 1. In fact, this bound holds for a large fraction ($\geq 1 - N^{-\alpha}$ for any fixed $\alpha \geq 1$) of all possible inputs.*

The above integer sorting algorithm can be used to randomly permute N given keys such that each permutation is equally likely. The idea is to assign a random label with each key in the range $[1, N^{1+\beta}]$ (for any fixed $0 < \beta < 1$) and sort the keys with respect to their labels. When the key labels are in the range $[1, N^{1+\beta}]$, the labels may not be unique. The maximum number of times any label is repeated is $\widetilde{O}(1)$. Keys with equal labels are permuted in one more pass through the data.

The following Theorem is proven in [11]:

Theorem 2. *We can permute N keys randomly in $O(\frac{\log(N/M)}{\log(M/B)})$ passes through the data with probability $\geq 1 - N^{-\alpha}$ for any fixed $\alpha \geq 1$, where μ is a constant < 1, provided $B = \Omega(\log N)$.*

3.2 Randomized Sorting

An asymptotically optimal randomized algorithm can be developed using the random permutation algorithm and merge sort [11]. A simple version of the algorithm (called RSort1) works as follows: 1) Randomly permute the input N keys; 2) In one pass through the data form runs each of length $M = DB$; 3) Use a D-way merge sort to merge the N/M runs. Let an *iteration* refer to the task of merging D runs. At the beginning of any iteration 2 blocks are brought in from each disk. The D runs are merged to ship DB keys out to the disks. From thereon, maintain the invariant that for each run there are two leading blocks in the memory. This means that after shipping DB keys to the disks, bring in enough keys for each run so that there will be two leading blocks per run.

Even though RSort1 makes an optimal number of scans through the input, each scan may take more than an optimal number of I/Os. This is because the runs get consumed at different rates. For instance, there could come a time when we need a block from each run and all of these blocks are in the same disk. At the beginning of any iteration, each run is striped across the disks one block per disk. Also, the leading blocks of successive runs will be in successive disks. For example, the leading block of run 1 could be in disk i, the leading block of run 2 could be in disk $i + 1$, etc. If this property could be maintained always, we can get perfect parallelism in reading the blocks. If the blocks are consumed at the same rate then this property will be maintained.

RSort1 is modified in [11] to get RSort2. In RSort2 leading blocks of successive runs are either in successive disks or very nearly so. When the leading blocks of successive runs deviate significantly in terms of their disk locations, a *rearrangement* operation is performed. This operation involves rearranging the leading DB keys of each run so that the above property is reinstated after the rearrangement. As time progresses, the leading blocks will deviate more and more from their expected disk locations. Call the step of bringing in enough keys to have 2 blocks per run, merging them and outputting DB keys as a *stage* of the algorithm. After every D stages RSort2 performs a rearrangement of runs.

Theorem 3. RSort2 *takes* $\widetilde{O}\left(\frac{\log(N/M)}{\log(M/B)}\right)$ *read passes through the data provided* $B = \Omega(\sqrt{M}\log N)$.

RSort2 assumes that $B = \Omega(\sqrt{M}\log N)$. This assumption is relaxed using a value of $R = D^{\epsilon}$ for any constant $1 > \epsilon > 0$. In the resultant algorithm RSort, rearrangements are done every D^{ϵ} stages.

For choice of $\epsilon = 1/2$, the number of iterations made by the algorithm is no more than $2(1+\nu)\cdot\frac{\log(N/M)}{\log(M/B)}$ where each iteration involves $2(1+\nu)$ read passes including rearrangement. The number of write passes is the same. This gives a total of $8(1+\nu)\frac{\log(N/M)}{\log(M/B)}$ passes with probability $\geq (1 - N^{-\alpha})$ for any constant $\alpha \geq 1$. Here ν, μ are constants between 0 and 1. To this, we must add the time for generating the initial random permutation.

When B is large, we can decrease the number of read passes made by RSort [11]. For instance if $B = M^{\beta}$, the number of read passes made by RSort is $(4-4\beta)(1+\nu)\frac{\log(N/M)}{\log(M/B)} + (1+\mu)\frac{\log(N/M)}{\log(M/B)} + 2$.

4 New Ideas

In Algorithm RSort2, a rearrangement is done every $R = D$ stages. This rearrangement involves reading the leading DB keys of each run and writing them back so that the leading blocks of the runs are in successive disks. One of the key ideas is not to do the rearrangements. Instead remove leading relevant keys from each run so that after removing these keys, the leading keys of each run start from successive disks. The removed keys are stored in the disks as a *dirty sequence*. After the algorithm completes execution in this fashion, there will be a long sorted sequence and a dirty sequence. The dirty sequence is sorted and merged with the long sorted sequence. We will show that the length of the dirty sequence is much smaller than the length of the main sorted sequence and hence the final merging step does not take much time. Call this algorithm RSort3.

First we consider the problem of merging a sorted sequence X of length n and an unsorted sequence Y of length m. First we can sort Y using any of the optimal algorithms. This can be done in $O\left(\frac{m}{DB}\frac{\log(m/M)}{\log(M/B)}\right)$ parallel I/O operations. Let the sorted order of Y be Z. We can merge X and Z as follows. Assume that X is striped starting from disk 1 and Z is striped starting from disk $(D/2)+1$. We can use a simple merging algorithm where at the beginning we bring D blocks from each run. Get the DB smallest keys out of these and ship them out to the disks. Now bring in enough keys from each run so that there are DB keys from each run. Get the DB smallest keys from these and ship them out, and so on. For every DB keys output, we have to perform at most two parallel read I/O operations. Thus the total number of parallel read I/O's needed to merge X and Z is $\leq \frac{2(m+n)}{DB}$. As a result, we infer the following:

Theorem 4. *We can merge a sorted sequence X with an unsorted sequence Y in $O\left(\frac{m}{DB}\frac{\log(m/M)}{\log(M/B)}\right)+\frac{2(m+n)}{DB}$ parallel read I/O operations.*

Length of the Dirty Sequence: Note that a rerrangement is done in RSort2 every R stages, i.e., for every RDB keys output. The contribution to the length of the dirty sequence by every R stages is at most three blocks from every run (with high probability). In other words, the length of the dirty sequence increases by $3RB$ for every RDB keys output. This means that the length of the dirty sequence during the entire algorithm is no more than $(1+\nu)\frac{\log(N/DB)}{\log(D)} \times N \times \frac{3RB}{RDB} = (1+\nu)\frac{\log(N/DB)}{\log(D)} \times \frac{3N}{D}$ with high probability (where ν is any constant > 0 and < 1).

Following the analysis of RSort2, we get the following:

Theorem 5. RSort3 *is asymptotically optimal and since it eliminates the rearrangement step completely, the underlying constant is smaller than that of* RSort2. *For example, when $\beta = 1/2$, the constant reduces roughly by a factor of 40%.*

The case of $R = D^\epsilon$: The constraint on B is relaxed in [11] by choosing a value of $R = D^\epsilon$ (for some constant $\epsilon > 0$). For this version (called RSort) of the algorithm, a rearrangement is done every D^ϵ stages. The total number of keys output in these many stages is DBD^ϵ. The corresponding contribution to the length of the dirty sequence is $(DB/R)D^\epsilon$. Thus the length of the dirty sequence in the entire algorithm does not exceed $\frac{(1+\nu)}{\epsilon}\frac{\log(N/DB)}{\log(D)}\frac{N}{D^\epsilon}$ with high probability.

When $\epsilon = 1/2$, the length of the dirty sequence is no more than $2(1+\nu)\frac{\log(N/DB)}{\log(D)}\frac{N}{\sqrt{D}}$ with high probability (for any constant $\nu > 0$). In this case the initial permutation takes $(1+\mu)\frac{\log(N/DB)}{\log(D)}$ passes through the data (for any constant $\mu > 0$). Initial runs can be formed in one pass. Merging of runs takes $2\frac{(1+\nu)}{\epsilon}\frac{\log(N/DB)}{\log(D)}$ read passes through the data. If we neglect the time taken to process and merge the dirty sequence, then the number of passes taken by RSort3 is roughly 40% less than that of RSort.

5 Implementation and Experimental Details

We have simulated the PDM on a single disk computer with a **focus on counting the number of parallel I/Os needed**. Random data has been employed. We have compared the number of parallel I/Os with that of the lower bound of $\frac{N}{DB}\frac{\log\frac{N}{DB}}{\log(D)}$. **Please note that this way of measuring the performance of a PDM algorithm is preferable since the results obtained could be easily translated onto any other implementation (hardware or otherwise) of a PDM**.

A simple way of simulating a PDM with a single disk is to have a file corresponding to each disk. If we are only interested in counting the number of parallel I/O operations needed, the detail on how the disks are implemented is immaterial.

The specific algorithm we have implemented works as follows: In one pass through the data we form initial runs of length DB each. We employ a D-way merge to merge the runs. This in particular means that there will be $\frac{\log(N/DB)}{\log D}$ *levels* of merging (such that in each level we reduce the number of runs by a factor of D). In each level there will be many *iterations*. In each iteration we merge D runs into one.

Consider any iteration of the algorithm where we merge the runs $R_1, R_2, \ldots, R_D$. At the beginning of the iteration, the runs will be stored in the disks as follows. Let the leading block of R_1 be in disk i (for some $0 \leq i \leq (D-1)$). The second block of R_1 will be in disk $(i+1)$ mod D, the third block of R_1 will be in disk $(i+2)$ mod D, and so on. Also, the leading block of R_2 will be in disk $(i+1)$ mod D, the second block of R_2 will be in disk $(i+2)$ mod D, etc.

At any time during an iteration of the algorithm, we keep two leading blocks per run in the main memory. In particular, we begin any iteration by bringing in two leading blocks per run. We identify the smallest DB keys from the D runs. It can be shown (using Chernoff bounds) that, with high probability, we will not have to bring in any more keys from the disks to produce these DB smallest keys. These DB keys are shipped to the disks in one parallel I/O operation. After this, we examine the runs in the memory and refill the runs so that we will have two leading blocks per run. This refill operation involves reading from the disks enough keys per run so that we will have the required two blocks per run.

The above process is repeated until the D runs are merged. An important question is how many I/O operations will be needed to do the refill operation. In the ideal case each run will be consumed at exactly the same rate. In this case the refill operation can be done in one parallel I/O operation. But this may not be the typical case. The runs may be consumed at different rates and hence we may have to read multiple blocks from each disk. Therefore, for each refill operation we figure out the maximum number of blocks that we may have to read in from any disk and charge the refill operation with these many I/Os. All the I/O counts reported in this paper are based on this scheme.

One can employ either the rearrangement scheme of [11] or the dirty sequence idea proposed in this paper to modify the algorithm. For example, when the maximum number of blocks to be fetched from any disk in a refill operation exceeds a threshold value (e.g., 3) we can either do a rearrangement or clean up with the creation of a dirty sequence.

5.1 Simulating the PDM Using a Single Disk

We use a single disk machine to simulate the PDM model. We start with a single binary input file (key_file) consisting of integers and create runs each of size DB and put them in a file called run_file. Every run in the run_file is preceded by a 4-byte unsigned long which is the size of the run following it. As described in the algorithm the runs are striped across the disks and a parallel read or write operation would read or write DB keys. In this section we give the details of how we simulated the parallel I/O operations. Every run i in the run_file has an offset pointer run_offset[i] which tells us at which position the next block of the run starts in the run_file. This section assumes that the readers are familiar with UNIX lseek, read and write system calls. Please see [17] for details of UNIX system calls.

5.2 Counting Parallel I/O's

A parallel I/O (for D disks) read operation can be simulated as follows. The following operations show how to initialize the run_offset[i] values of the consecutive runs. Assume that the file descriptor for the run_file is $runfd$.

- lseek(runfd,(off_t)0,SEEK_SET).
- read(runfd,&run_size,sizeof(unsigned long).
- run_offset[i] = lseek(runfd, (off_t)0, SEEK_CUR).

The above steps will read the length of the run and keep track of its offset for the next read of the block. Once we read the current run its length is stored in run_size. We can use this to read the next run as follows.

- lseek(runfd,(off_t)run_size,SEEK_CUR)
- read(runfd,&run_size,sizeof(unsigned long))
- run_offset[i + 1] = lseek(runfd, (off_t)0, SEEK_CUR).

The above steps indicate how to fill run_offset[i] for each run i. Now we show how we can do a parallel read. The following steps basically position the disk head corresponding to each run at run_offset[i] and reads a block of size B and updates the run_offset[i] so that the next read can get the next block. The block size (page size) on a 32-bit UNIX machine is $B = 4096$.

- lseek(runfd,(off_t)run_offset[i],SEEK_SET)
- read(runfd,(void *)buf,4096)
- run_offset[i] = lseek(runfd, (off_t)0, SEEK_CUR)

With this background on how to simulate the PDM model on a single disk we describe the algorithm based on this framework in the next section.

5.3 Algorithm Implementation

Algorithm 1 gives an outline of the ideas. We start off with $\frac{N}{DB}$ runs and merge D runs each time and during the merge we handle two situations. In the first situation during the D-way merge we may be able to fill upto DB keys without

Algorithm 1. Outline of our PDM sorting algorithm

```
INPUT   : A Binary file of integers, parameters $N$ and $D$
OUTPUT: A Binary file of sorted integers
Scan through the data once to form initial runs of length $DB$ each.
// Keep merging until you get a single run
while $runs > 1$ do
    Merge the next $D$ runs as follows:
    Let $R_1, R_2, \ldots, R_D$ be these runs
    for $i = 1;i \leq D;i++$ do
        Bring two leading blocks from $R_i$;
    while the runs are not merged do
        Get the smallest $DB$ keys from the $D$ runs;
        If keys from the disks are needed to get these
        $DB$ smallest keys, quit and start all over;
        Ship the smallest $DB$ keys to the disks in one I/O;
        These keys will add to the run corresponding to the
        merge of $R_1, R_2, \ldots, R_D$;
        for $i = 1;i \leq D;i++$ do
            Bring enough keys from $R_i$ so that there will be
            two leading blocks of $R_i$ in memory;
```

Algorithm 2. GetMaxParIO computes parallel I/O's to refill

```
INPUT   : runheads of the $D$ runs
OUTPUT: Parallel I/O's to refill
//Find the disk number of leading block
//of each run
for $i = 0$ to $D$ do
    $disk\_number[k] = (\lceil \frac{runhead[i]}{B} \rceil + k)mod(D)$ ;
Sort($disk\_number$) ;
$temp\_par\_io \leftarrow 1$ ;
$max\_par\_io \leftarrow 1$ ;
for $i = 1$ to $D$ do
    if $disk\_number[i-1] == disk\_number[i]$ then
        $temp\_par\_io = temp\_par\_io + 1$ ;
    else
        if $temp\_par\_io > max\_par\_io$ then
            $max\_par\_io = temp\_par\_io$ ;
        $temp\_par\_io = 1$ ;
return $max\_par\_io$ ;
```

Table 1. Parallel I/0's for D=4

N	$\frac{N}{DB}\frac{log(\frac{N}{DB})}{log(D)}$	Our Algo	$\frac{OurAlgo}{Lowerbound}$
2^{16}	32	76	2.3750
2^{18}	192	414	2.1562
2^{20}	1024	2102	2.0527
2^{22}	5120	10136	1.9797
2^{24}	24576	48404	1.9696
2^{26}	114688	221257	1.9292
2^{28}	524288	995356	1.8985
2^{28}	524288	995356	1.8985

Table 2. Parallel I/0's for D=8

N	$\frac{N}{DB}\frac{log(\frac{N}{DB})}{log(D)}$	Our Algo	$\frac{OurAlgo}{Lowerbound}$
2^{19}	128	324	2.5312
2^{22}	1536	3579	2.3301
2^{25}	16384	37562	2.2926
2^{28}	163840	388585	2.3717
2^{28}	163840	388585	2.3717

Table 3. Parallel I/0's for D=16

N	$\frac{N}{DB}\frac{log(\frac{N}{DB})}{log(D)}$	Our Algo	$\frac{OurAlgo}{Lowerbound}$
2^{22}	512	1287	2.5137
2^{26}	12288	30200	2.4577

Table 4. PARALLEL I/0 frequency D=4

N	Total I/0's	Max(in 1 refill)	Frequency
2^{20}	1778	2	[1]=610 [2]=584
2^{22}	8948	2	[1]=2656 [2]=3146
2^{24}	42881	3	[1]=11742 [2]=15553 [3]=11
2^{26}	199800	3	[1]=52428 [2]=72174 [3]=1008
2^{28}	915091	3	[1]=226981 [2]=334881 [3]=6116

running out of keys from the merge buffers of the D runs. This will successfully complete a *refill* step. The *refill* steps contribute to the bulk of the parallel I/O's because the disk heads may not move uniformly. To compute the parallel I/O's

Table 5. PARALLEL I/0 frequency D=8

N	Total I/0's	Max(in 1 refill)	Frequency
2^{19}	241	2	[1]=51 [2]=95
2^{22}	2978	2	[1]=386 [2]=1296
2^{25}	32323	4	[1]=3233 [2]=13874 [3]=446 [4]=1
2^{28}	343753	5	[1]=26319 [2]=126770 [3]=16830[4]=3011 [5]=272

Table 6. PARALLEL I/0 frequency D=16

N	Total I/0's	Max(in 1 refill)	Frequency
2^{22}	1007	2	[1]=85 [2]=461
2^{26}	25254	4	[1]=1371 [2]=10514 [3]=941 [4]=8

required to *refill* we keep track of how much of a run is consumed (or read into main memory) in a variable called runhead. So runhead[i] at any stage of the D-way merge indicates how much of run i has been consumed. This runhead information is passed to the parallel I/O computation algorithm in Algorithm 2. This algorithm finds for each run in which disk the leading block of the run is found and then sorts these numbers and finds the maximum number of times a disk is repeated. This gives the maximum number of parallel I/O's needed for the refill operation.

5.4 Program Download and Tests

The program implementing our idea can be downloaded from `http://trinity.engr.uconn.edu/ vamsik/PDMSorting/PDMSorting.tgz`.

6 Conclusions

In this paper we have presented simple algorithms that are asymptotically optimal. We have implemented these algorithms and the number of I/O operations is very close to the lower bound indicating that our algorithms are practical.

References

1. Aggarwal, A., Plaxton, G.: Optimal parallel sorting in multi-level storage. In: Proc. of the ACM-SIAM SODA 1994, pp. 659–668 (1994)
2. Aggarwal, A., Vitter, J.S.: The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM 31(9), 1116–1127 (1988)

3. Arge, L.: The Buffer Tree: A New Technique for Optimal I/O-Algorithms. In: Sack, J.-R., Akl, S.G., Dehne, F., Santoro, N. (eds.) WADS 1995. LNCS, vol. 955, pp. 334–345. Springer, Heidelberg (1995)
4. Barve, R., Grove, E.F., Vitter, J.S.: Simple Randomized Mergesort on Parallel Disks. Parallel Computing 23(4-5), 601–631 (1997)
5. Dementiev, R., Sanders, P.: Asynchronous Parallel Disk Sorting. In: Proc. ACM Symposium on Parallel Algorithms and Architectures, pp. 138–148 (2003)
6. Horowitz, E., Sahni, S., Rajasekaran, S.: Computer Algorithms. W. H. Freeman Press, New York (1998)
7. Hutchinson, D., Sanders, P., Vitter, J.: Duality between prefetching and queued writing with parallel disks. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, pp. 62–73. Springer, Heidelberg (2001)
8. Nodine, M., Vitter, J.: Deterministic distribution sort in shared and distributed memory multirocessors. In: Proc. of the ACM SPAA 1993, pp. 120–129 (1993), `http://www.cs.duke.edu/~jsv/Papers/catalog/node16.html`
9. Nodine, M.H., Vitter, J.S.: Greed Sort: Optimal Deterministic Sorting on Parallel Disks. Journal of the ACM 42(4), 919–933 (1995)
10. Rajasekaran, S.: A Framework for Simple Sorting Algorithms on Parallel Disk Systems. Theory of Computing Systems 34(2), 101–114 (2001)
11. Rajasekaran, S., Sen, S.: Optimal and Practical Algorithms for Sorting on the PDM. IEEE Transactions on Computers 57(4) (2008)
12. Rajasekaran, S., Sen, S.: PDM Sorting Algorithms That Take A Small Number Of Passes. In: Proc. International Parallel and Distributed Processing Symposium (IPDPS) (2005)
13. Rajasekaran, S., Sen, S.: A Simple Optimal Randomized Sorting Algorithm for the PDM. In: Deng, X., Du, D.-Z. (eds.) ISAAC 2005. LNCS, vol. 3827, pp. 543–552. Springer, Heidelberg (2005)
14. Sanders, P., Enger, S., Korst, J.: Fast concurrent access to parallel disks. In: Proc. of the ACM-SIAM SODA 2000, pp. 849–858 (2000)
15. Vitter, J.S., Hutchinson, D.A.: Distribution Sort with Randomized Cycling. In: Proc. 12th Annual SIAM/ACM Symposium on Discrete Algorithms (2001)
16. Vitter, J.S., Shriver, E.A.M.: Algorithms for Parallel Memory I: Two-Level Memories. Algorithmica 12(2-3), 110–147 (1994)
17. Stevens, W.R., Rago, S.A.: Advanced Programming in the UNIX Environment, 2nd edn. Addison-Wesley Professional Computing Series (2007)

Appendix A: Chernoff Bounds

If a random variable X is the sum of n independent and identically distributed Bernoulli trials with a success probability of p in each trial, the following equations give us concentration bounds of deviation of X from the expected value of np. The first equation is more useful for large deviations whereas the other two are useful for small deviations from a large expected value.

$$Prob(X \geq m) \leq \left(\frac{np}{m}\right)^m e^{m-np} \tag{1}$$

$$Prob(X \leq (1-\epsilon)pn) \leq exp(-\epsilon^2 np/2) \tag{2}$$

$$Prob(X \geq (1+\epsilon)np) \leq exp(-\epsilon^2 np/3) \tag{3}$$

for all $0 < \epsilon < 1$.

Accelerating Cone Beam Reconstruction Using the CUDA-Enabled GPU[*]

Yusuke Okitsu, Fumihiko Ino, and Kenichi Hagihara

Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
{y-okitu,ino}@ist.osaka-u.ac.jp

Abstract. Compute unified device architecture (CUDA) is a software development platform that enables us to write and run general-purpose applications on the graphics processing unit (GPU). This paper presents a fast method for cone beam reconstruction using the CUDA-enabled GPU. The proposed method is accelerated by two techniques: (1) off-chip memory access reduction; and (2) memory latency hiding. We describe how these techniques can be incorporated into CUDA code. Experimental results show that the proposed method runs at 82% of the peak memory bandwidth, taking 5.6 seconds to reconstruct a 512^3-voxel volume from 360 512^2-pixel projections. This performance is 18% faster than the prior method. Some detailed analyses are also presented to understand how effectively the acceleration techniques increase the reconstruction performance of a naive method.

1 Introduction

Cone beam (CB) reconstruction is an imaging process for producing a three-dimensional (3-D) volume from a sequence of 2-D projections obtained by a CB computed tomography (CT) scan. This reconstruction technique is integrated into many mobile C-arm CT systems in order to assist the operator during image-guided surgery. In general, a CB reconstruction task should be completed within ten seconds because the operator has to stop the surgical procedure until obtaining the intraoperative volume. However, it takes 3.21 minutes to obtain a 512^3-voxel volume on a single 3.06 GHz Xeon processor [1]. Accordingly, many projects are trying to accelerate CB reconstruction using various accelerators, such as the graphics processing unit (GPU) [2,3,4,5,6], Cell [1], and FPGA [7].

To the best of our knowledge, Xu et al. [2] show the fastest method using the GPU, namely a commodity chip designed for acceleration of graphics tasks. Their method is implemented using the OpenGL library in order to take an advantage of graphics techniques such as early fragment kill (EFK). It takes 8.3 seconds to reconstruct a 512^3-voxel volume from 360 projections. In contrast to this graphics-based implementation strategy, a non-graphics implementation strategy is proposed by Scherl et al. [3]. They use compute unified device architecture (CUDA) [8] to implement CB reconstruction

[*] This work was partly supported by JSPS Grant-in-Aid for Scientific Research (A)(2) (20240002), Young Researchers (B)(19700061), and the Global COE Program "in silico medicine" at Osaka University.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 108–119, 2008.

on the GPU. The reconstruction of a 512^3-voxel volume from 414 projections takes 12.02 seconds, which is slightly longer than the graphics-based result [2]. However, it is still not clear whether the CUDA-based strategy will outperform the graphics-based strategy, because their implementation is not presented in detail. In particular, optimization techniques for CUDA programs are of great interest to the high-performance computing community.

In this paper, we propose a CUDA-based method capable of accelerating CB reconstruction on the CUDA-enabled GPU. Our method is based on the Feldkamp, Davis, and Kress (FDK) reconstruction algorithm [9], which is used in many prior projects [1,2,3,4,5,7,10]. We optimize the method using two acceleration techniques: (1) one is for reducing the number and amount of off-chip memory accesses; and (2) another for hiding the memory latency with independent computation. We also show how effectively these techniques contribute to higher performance, making it clear that the memory bandwidth and the instruction issue rate limit the performance of the proposed method.

2 Related Work

Xu et al. [2] propose an OpenGL-based method accelerated using the graphics pipeline in the GPU. They realize a load balancing scheme by moving instructions from fragment processors to vertex processors, each composing the pipeline. This code motion technique also contributes to reduce the computational complexity [11]. Furthermore, their method uses the EFK technique to restrict computation to voxels within the region of interest (ROI). Although this fragment culling technique leads to acceleration, we cannot obtain the correct data outside the ROI, where fragments are culled from rendering. In contrast, our goal is to achieve higher reconstruction performance for the entire volume.

Scherl et al. [3] show a CUDA-based method with a comparison to a Cell-based method. They claim that their method reduces the number of instructions and the usage of registers. In contrast, our acceleration techniques focus on reducing the number and amount of off-chip memory accesses and hiding the memory latency with computation. Such memory optimization is important to improve the performance of the FDK algorithm, which can be classified into a memory-intensive problem.

Another acceleration strategy is to perform optimization at the algorithm level. For example, the rebinning strategy [12] rearranges and interpolates CB projections to convert them into parallel beam projections. This geometry conversion simplifies the backprojection operation needed for the FDK reconstruction. One drawback of the rebinning strategy is that it creates artifacts in the final volume. Using this rebinning strategy, Li et al. [10] develop a fast backprojection method for CB reconstruction. Their method is implemented using CUDA and takes 3.6 seconds to perform backprojection of 360 512^3-pixel projections. In contrast, our method accelerates the entire FDK algorithm for CB projections without rebinning.

3 Overview of CUDA

Figure 1 illustrates an overview of the CUDA-enabled GPU. The GPU consists of multiprocessors (MPs), each having multiple stream processors (SPs). Each MP has small

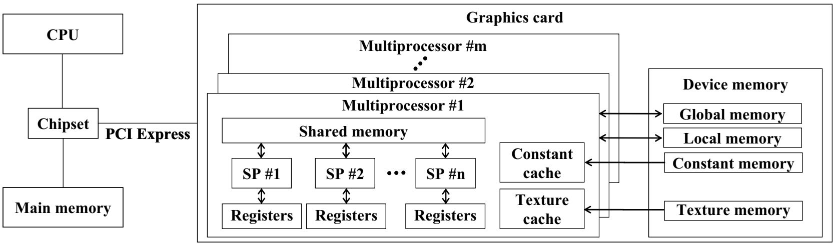


Fig. 1. Architecture of CUDA-enabled GPU. SP denotes a stream processor.

on-chip memory, called shared memory, which can be accessed from internal SPs as fast as registers. However, it is not shared between different MPs. Due to this constraint, threads are classified into groups and each group is called as a block, which is the minimum allocation unit assigned to an MP. That is, developers have to write their code such that there is no dependence between threads in different blocks. On the other hand, threads in the same block are allowed to have dependence because they can communicate each other by shared memory.

CUDA also exposes the memory hierarchy to developers, allowing them to maximize application performance by optimizing data access. As shown in Fig. 1, there is off-chip memory, called device memory, containing texture memory, constant memory, local memory, and global memory. Texture memory and constant memory have a cache mechanism but they are not writable from SPs. Therefore, developers are needed to transfer (download) data from main memory in advance of a kernel invocation. Texture memory differs from constant memory in that it provides a hardware mechanism that returns linearly interpolated texels from the surrounding texels. On the other hand, local memory and global memory are writable from SPs but they do not have a cache mechanism. Global memory achieves almost the full memory bandwidth if data accesses can be coalesced into a single access [8]. Local memory cannot be explicitly used by developers. This memory space is implicitly used by the CUDA compiler in order to avoid resource consumption. For example, an array will be allocated to such space if it is too large for register space. Local memory cannot be accessed in a coalesced manner, so that it is better to eliminate such inefficient accesses hidden in the kernel code.

4 Feldkamp Reconstruction

The FDK algorithm [9] consists of the filtering stage and the backprojection stage. Suppose that a series of $U \times V$-pixel projections $P_1, P_2, \dots P_K$ are obtained by a scan rotation of a detector in order to create an N^3-voxel volume F. The algorithm then applies the Shepp-Logan filter [13] to each projection, which gives a smoothing effect to minimize noise propagation at the backprojection stage. At this filtering stage, the pixel value $P_n(u, v)$ at point (u, v) is converted to value $Q_n(u, v)$ such that:

$$Q_n(u, v) = \sum_{s=-S}^{S} \frac{2}{\pi^2(1 - 4s^2)} W_1(s, v) P_n(s, v), \tag{1}$$

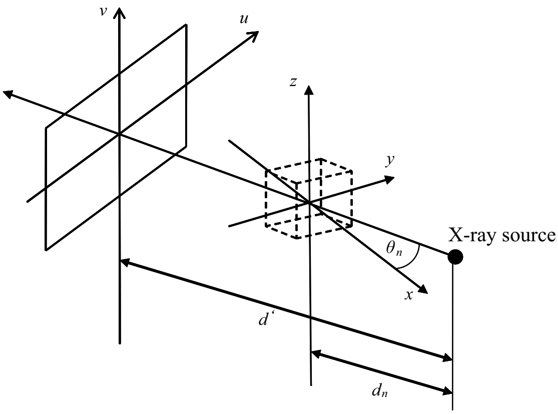


Fig. 2. Coordinate system for backprojection. The xyz space represents the volume while the uv plane represents a projection that is to be backprojected to the volume.

where S represents the filter size and $W_1(s, v)$ represents the where d' represents the distance between the X-ray source and the origin of the detector (projection), as shown in Fig. 2.

A series of filtered projections $Q_1, Q_2, \ldots Q_K$ are then backprojected to the volume F. In Fig. 2, the xyz space corresponds to the target volume F while the uv plane represents the n-th projection P_n that is to be backprojected to volume F from angle θ_n, where $1 \leq n \leq K$. Note here that the distance d_n between the X-ray source and the volume origin is parameterized for each projection, because it varies during the rotation of a real detector. On the other hand, distance d' can be modeled as a constant value in C-arm systems.

Using the coordinate system mentioned above, the voxel value $F(x, y, z)$ at point (x, y, z), where $0 \leq x, y, z \leq N - 1$, is computed by:

$$F(x, y, z) = \frac{1}{2\pi K} \sum_{n=1}^{K} W_2(x, y, n) Q_n(u(x, y, n), v(x, y, z, n)), \tag{2}$$

where the weight value $W_2(x, y, n)$, the coordinates $u(x, y, n)$ and $v(x, y, z, n)$ are given by:

$$W_2(x, y, n) = \left(\frac{d_n}{d_n - x cos\theta_n - y sin\theta_n} \right)^2, \tag{3}$$

$$u(x, y, n) = \frac{d'(-x sin\theta_n + y cos\theta_n)}{d_n - x cos\theta_n - y sin\theta_n}, \tag{4}$$

$$v(x, y, z, n) = \frac{d' z}{d_n - x cos\theta_n - y sin\theta_n}. \tag{5}$$

The coordinates $u(x, y, n)$ and $v(x, y, z, n)$ are usually real values rather than integer values. Since projections $P_1, P_2, \ldots P_K$ are given as discrete data, we need an interpolation mechanism to obtain high-quality volume.

5 Proposed Method

To make the description easier to understand, we first show a naive method and then the proposed method with acceleration techniques.

5.1 Parallelization Strategy

Since a 512^3-voxel volume requires at least 512 MB of memory space, it is not easy for commodity GPUs to store both the entire volume and the projections in device memory. To deal with this memory capacity problem, we have decided to store the entire volume in device memory because earlier projections can be processed and removed before the end of a scan rotation. In other words, this decision allows us to structure the reconstruction procedure into a pipeline. Figure 3 shows an overview of our reconstruction method. In the naive method, the first projection P_1 is transferred to global memory, which is then filtered and backprojected to the volume F in global memory. This operation is iteratively applied to the remaining projections to obtain the final accumulated volume F. See also Fig. 4 for the pseudocode of the naive method.

In Fig. 4, the filtering stage is parallelized in the following way. Eq. (1) means that this stage performs a 1-D convolution in the u-axis direction. Thus, there is no data dependence between different pixels in a filtered projection Q_n. However, pixels in the same column u refer partly the same pixels in projection P_n. Therefore, it is better to use shared memory to save the memory bandwidth. Thus, we have decided to write the filtering kernel such that a thread block is responsible for applying the filtering operation to pixels in a column. On the other hand, a thread is responsible for computing a pixel value $Q_n(u, v)$. As shown in Fig. 4, threads in the same block cooperatively copy a column u to shared memory at line 13, which are then accessed instead of the original data in global memory at line 15.

Similarly, there is no constraint at the backprojection stage in terms of parallelism. That is, any voxel can be processed at the same time. However, it is better to use 1-D or 2-D thread blocks rather than 3-D thread blocks in order to reduce the computational complexity by data reuse. This data reuse technique can be explained by Eqs. (3) and

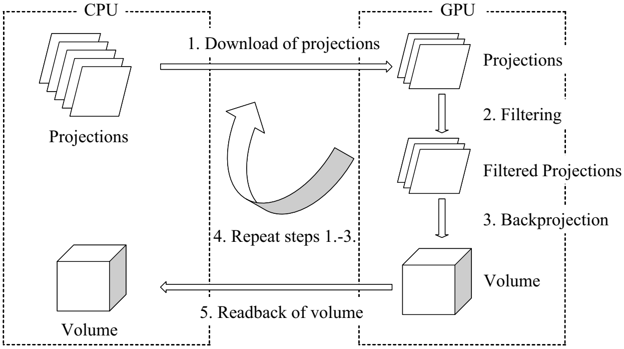


Fig. 3. Overview of the proposed method. Projections are serially sent to the GPU in order to accumulate their pixels into the volume in video memory.

```
Input: Projections $P_1 \dots P_K$, filter size $S$ and
        parameters $d', d_1 \dots d_K, \theta_1 \dots \theta_K$
Output: Volume $F$

Algorithm NaiveReconstruction()
 1: Initialize volume F
 2: for $n = 1$ to $K$ do
 3:     Transfer projection $P_n$ to global memory
 4:     $Q \leftarrow$ FilteringKernel($P_n, S$)
 5:     Bind filtered projection $Q$ as a texture
 6:     $F \leftarrow$ BackprojectionKernel($Q, d', d_n, \theta_n, n$)
 7: end for
 8: Transfer volume $F$ to main memory

Function FilteringKernel($P, S$)
 9: __shared__ float $array[U]$        // $U$: projection width
10: $u \leftarrow \mathrm{index}(\mathrm{threadID})$    // returns responsible $u$
11: $v \leftarrow \mathrm{index}(\mathrm{blockID})$
12: Initialize $Q(u, v)$
13: $array[u] \leftarrow W_1(u, v) * P(u, v) * 2/\pi^2$
14: for $s = -S$ to $S$ do
15:     $Q(u, v) \leftarrow Q(u, v) + array[u + s]/(1 - 4s^2)$
16: end for

Function BackprojectionKernel($Q, d', d_n, \theta_n, n$)
17: $x \leftarrow \mathrm{index}(\mathrm{blockID})$
18: $y \leftarrow \mathrm{index}(\mathrm{threadID})$
19: $u \leftarrow u(x, y, n)$             // Eq. (4)
20: $v \leftarrow v(x, y, 0, n)$          // Eq. (5)
21: $v' \leftarrow v'(x, y, n)$           // Eq. (6)
22: for $z = 0$ to $N - 1$ do
23:     $F(x, y, z) \leftarrow F(x, y, z) + W_2(x, y, n) * Q(u, v)$
24:     $v \leftarrow v + v'$
25: end for
```

Fig. 4. Pseudocode of the native method. This code is a simplified version.

(4), which indicate that $W_2(x, y, n)$ and $u(x, y, n)$ do not depend on z. Therefore, these two values can be reused for voxels in a straight line along the z-axis: line $(X, Y, 0) - (X, Y, N - 1)$, where X and Y are constant values in the range $[0, N - 1]$. To perform this data reuse, our naive method employs 1-D thread blocks (but 2-D blocks after optimization shown later in Section 5.2) that assign such voxels to the same thread. In summary, a thread is responsible for a line while a thread block is responsible for a set of lines: plane $x = X$ for thread block X, where $0 \leq X \leq N - 1$.

The data reuse can be further applied to reduce the complexity of Eq. (5). Although $v(x, y, z, n)$ depends on z, it can be rewritten as $v(x, y, z, n) = v'(x, y, n)z$, where

$$v'(x, y, n) = \frac{d'}{d_n - xcos\theta_n - ysin\theta_n}. \qquad (6)$$

Therefore, we can precompute $v'(x, y, n)$ for any z (line 21), in order to incrementally compute Eq. (5) at line 24.

```
Function OptimizedBackprojectionKernel($Q[I * J], d', d_n[I * J], \theta_n[I * J], n$)
var $u[I], v[I], v'[I], w[I]$
$x \leftarrow \mathrm{index(blockID, threadID)}$
$y \leftarrow \mathrm{index(blockID, threadID)}$
for $j = 0$ to $J - 1$ do        // unrolled
    for $i = 0$ to $I - 1$ do
        $w[i] \leftarrow W_2(x, y, 3j + i + n)$        // Eq. (3)
        $u[i] \leftarrow u(x, y, 3j + i + n)$          // Eq. (4)
        $v[i] \leftarrow v(x, y, 0, 3j + i + n)$       // Eq. (5)
        $v'[i] \leftarrow v'(x, y, 3j + i + n)$        // Eq. (6)
    end for
    for $z = 0$ to $N - 1$ do
        $F(x, y, z) \leftarrow F(x, y, z) + w[0] * Q[3j](u[0], v[0])$
                                $+ w[1] * Q[3j + 1](u[1], v[1])$
                                $\ldots$
                                $+ w[I - 1] * Q[3j + (I - 1)](u[I - 1], v[I - 1])$
        for $k = 0$ to $I - 1$ do
            $v[k] \leftarrow v[k] + v'[k]$
        end for
    end for
end for
```

Fig. 5. Pseudocode of the proposed method. The actual code is optimized by loop unrolling, for example.

Note that the filtered projection data is accessed as a texture. As we mentioned in Section 4, the coordinates $u(x, y, n)$ and $v(x, y, z, n)$ are usually real values. Therefore, we load the data $Q_n(u, v)$ from a texture, which returns a texel value interpolated by hardware. This strategy contributes to a full utilization of the GPU, because the interpolation hardware is separated from processing units.

5.2 Accelerated Backprojection

The acceleration techniques we propose in this paper optimize the backprojection kernel of the naive method. These techniques are motivated to maximize the effective memory bandwidth because the backprojection stage is a memory-intensive operation. We maximize the effective bandwidth by two techniques which we mentioned in Section 1. The naive method presented in Fig. 4 is modified to the optimized code shown in Fig. 5 by the following five steps.

1. Memory access coalescing [8]. This technique is important to achieve a full utilization of the wide memory bus available in the GPU. We store the volume data in global memory so that the memory accesses can be coalesced into a single contiguous, aligned memory access. This can be realized by employing 2-D thread blocks instead of 1-D blocks. It also improves the locality of texture access, which leads to a higher utilization of the texture cache.
2. Global memory access reduction. We modify the kernel to perform backprojection of I projections at a time, where I represents the number of projections processed by

a single kernel invocation. This modification reduces the number of global memory accesses to $1/I$ because it allows us to write temporal voxel values to local memory before writing the final values to global memory. We cannot use registers because a large array of size N is needed to store the temporal values for all z (line 23 in Fig. 4). Note that the increase of I results in more consumption of resources such as registers. We currently use $I = 3$, which is experimentally determined for the target GPU.

3. Local memory access reduction. The technique mentioned above decreases accesses to global memory but increases those to local memory. In order to reduce them, we pack I successive assignments into a single assignment. This modification is useful if the assignments have the same destination variable placed in local memory.
4. Local memory access elimination. We now have a single assignment for accumulation, so that we can write the accumulated values directly to global memory, as shown at line 12 in Fig. 5.
5. Memory latency hiding. We pack J successive kernel calls into a single call by unrolling the kernel code. A kernel invocation now processes every I projections J times. This modification is useful to hide the memory latency with computation. For example, if SPs are waiting for memory accesses needed for the first I projections, they can perform computation for the remaining $I(J-1)$ projections. As we did for I, we have experimentally decided to use $J = 2$.

6 Experimental Results

In order to evaluate the performance of the proposed method, we now show some experimental results including a breakdown analysis of execution time and a comparison with prior methods: the OpenGL-based method [2]; the prior CUDA-based method [3]; the Cell-based method [1]; and the CPU-based method [1]. For experiments, we use a desktop PC equipped with a Core 2 Duo E6850 CPU, 4GB main memory, and an nVIDIA GeForce 8800 GTX GPU with 768MB video memory. Our implementation runs on Windows XP with CUDA 1.1 and ForceWare graphics driver 169.21. Figure 6 shows the Shepp-Logan phantom [13], namely a standard phantom widely used for evaluation. The data size is given by $U = V = N = 512$, $K = 360$, and $S = 256$.

6.1 Performance Comparison

Table 1 shows the execution time needed for reconstruction of the Shepp-Logan phantom. Since the number K of projections differs from prior results [1,3], we have normalized them to the same condition ($K = 360$ and $U = V = 512$) as prior work [1,2] did in the paper. The proposed method achieves the fastest time of 5.6 seconds, which is 29% and 18% faster than the OpenGL-based method [2] and the prior CUDA-based method [3], respectively. This performance is equivalent to 64.3 projections per second (pps), which represents the throughput in terms of input projections. On the other hand, the image acquisition speed in recent CT scans ranges from 30 to 50 pps [2]. Therefore, the performance achieved by the proposed method is sufficient enough to produce the entire volume immediately after a scan rotation. Note here that the OpenGL-based method is also faster than the image acquisition speed if it is accelerated by the EFK

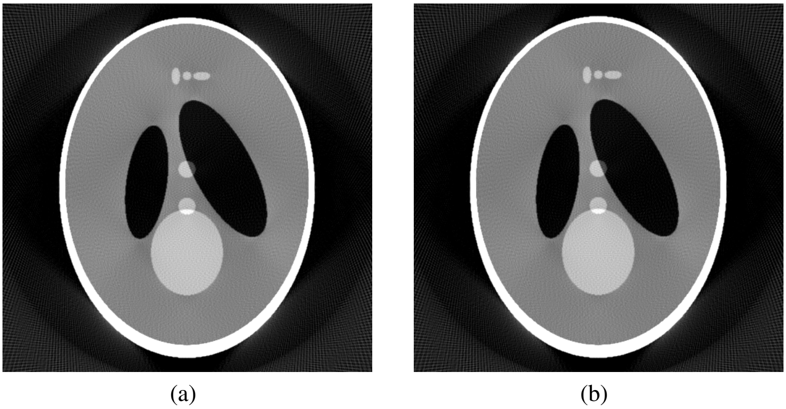


Fig. 6. Sectional views of the Shepp-Logan phantom [13] reconstructed (a) by the GPU and (b) by the CPU

Table 1. Performance comparison with prior methods. Throughput is presented by projections per second (pps).

Method	Hardware	Execution time (s)	Throughput (pps)
CPU [1]	Xeon 3.06 GHz	135.4	2.8
Cell [1]	Cell Broadband Engine	9.6	37.6
OpenGL [2]	GeForce 8800 GTX	8.9	40.5
Prior CUDA [3]	GeForce 8800 GTX	7.9	45.5
OpenGL w/ EFK [2]	GeForce 8800 GTX	6.8	52.9
Proposed method	GeForce 8800 GTX	5.6	64.3

technique. However, as we mentioned in Section 2, this technique does not reconstruct the volume area outside the ROI. In contrast, the proposed method reconstructs the entire volume within a shorter time.

Table 2 shows a breakdown of execution time comparing our method with the prior CUDA-based method. We can see that the acceleration is mainly achieved at the back-projection stage. As compared with the prior method, our method processes multiple projections at a kernel invocation. Therefore, we can reduce the number and amount of global memory accesses by packing I assignments into a single assignment, as shown at line 12 in Fig. 5. This reduction technique cannot be applied to the prior method, which processes a single projection at a time. Since we use $I = 3$, the proposed method achieves 67% less data transfer between MPs and global memory. With respect to the filtering stage, our method achieves the same performance as the prior method, which uses the nVIDIA CUFFT library. In this sense, we think that our filtering kernel achieves performance competitive to the vendor library. We also can see that the proposed method transfers the volume three times faster than the prior method. We think that this is due to the machine employed for the prior results, because the transfer rate is mainly determined by the chipset in the machine. Actually, there is no significant difference between the download rate and the readback rate in our method.

Table 2. Breakdown of execution time

Breakdown item	Proposed method (s)	Prior CUDA [3] (s)
Initialization	0.1	N/A
Projection download	0.2	0.2
Filtering	0.7	0.7
Backprojection	4.3	6.1
Volume readback	0.3	0.9
Total	5.6	7.9

Table 3. Effective floating point performance and memory bandwidth of our kernels. We assume that the GPU issues a single instruction per clock cycle and a stream processor executes two floating point (multiply-add) arithmetics per clock cycle. The effective memory bandwidth can be higher than the theoretical value due to cache effects.

Performance measure		Measured value		Theoretical value
		Filtering	Backprojection	
Instruction issue (MIPS)		1391	980	1350
Floating point (GFLOPS)	Processing units	105.8	38.3	345.6
	Texture units	—	124.3	172.8
	Total	105.8	162.6	518.4
Memory bandwidth (GB/s)		130.5	71.0	86.4

Table 3 shows the measured performance with the theoretical peak performance. We count the number of instructions in assembly code to obtain the measured values. This table indicates that the instruction issue rate limits the performance of the filtering kernel. Due to this bottleneck, the floating point performance results in 105.8 GFLOPS, which is equivalent to 20% of the peak performance. On the other hand, the effective memory bandwidth reaches 130.5 GB/s, which is higher than the theoretical value. This is due to the cache mechanism working for constant memory. The filtering kernel accesses 130 times more constant data, as compared with the variable data in global memory.

In contrast, the memory bandwidth is a performance bottleneck in the backprojection kernel. This kernel has more data access to global memory, which does not have cache effects. Actually, global memory is used for 40% of total amount. Thus, the backprojection kernel has lower effective bandwidth than the filtering kernel. However, the backprojection kernel achieves higher floating point performance because it exploits texture units for linear interpolation. The effective performance reaches 162.6 GFLOPS including 124.3 GFLOPS observed at texture units. Exploiting this hardware interpolation is important (1) to reduce the amount of data accesses between device memory and SPs and (2) to offload workloads from SPs to texture units. For example, SPs must fetch four times more texel data if we perform linear interpolation on them.

6.2 Breakdown Analysis

In order to clarify how each acceleration technique contributes to higher performance, we develop five subset implementations and measure their performance. Table 4 shows

Table 4. Backprojection performance with different acceleration techniques

Technique	Method					
	Naive	#1	#2	#3	#4	Proposed
1. Memory access coalescing	×	○	○	○	○	○
2. Global memory access reduction	×	×	○	○	○	○
3. Local memory access reduction	×	×	×	○	○	○
4. Local memory access elimination	×	×	×	×	○	○
5. Memory latency hiding	×	×	×	×	×	○
Backprojection time (s)	436.7	27.0	15.6	13.5	5.7	4.3

the details of each implementation with the measured time needed for backprojection of the Shepp-Logan phantom. Although the naive method is slower than the CPU-based method, the acceleration techniques reduce the backprojection time to approximately 1/102. This improvement is mainly achieved by memory access coalescing that reduces backprojection time to 27.0 seconds with a speedup of 16.2. In the naive method, every thread simultaneously accesses voxels located on the same coordinate x. This access pattern is the worst case, where 16 accesses can be coalesced into a single access [8], explaining why memory access coalescing gives such a speedup. Thus, the coalescing technique is essential to run the GPU as an accelerator for the CPU.

Reducing off-chip memory accesses further accelerates the backprojection kernel. As compared with method #1 in Table 4, method #4 has 66% less access to local memory and global memory, leading to 44% reduction of device memory access. On the other hand, the backprojection time is reduced to 5.7 seconds with a speedup of 4.7, whereas the speedup estimated from the reduction ratio of 44% becomes approximately 1.8. Thus, there is a gap between the measured speedup and the estimated speedup. We think that this gap can be explained by cache effects.

The last optimization technique, namely memory latency hiding, reduces the time by 25%. We analyze the assembly code to explain this reduction. Since we use $J = 2$ for the proposed method, we think that memory accesses for $j = 0$ can be overlapped with computation for $j = 1$ (line 4 in Fig. 5). We find that such overlapping computation takes approximately 1.3 seconds under the optimal condition, where MPs execute an instruction on each clock cycle. This probably explains why the time is reduced from 5.7 to 4.3 seconds.

7 Conclusion

We have presented a fast method for CB reconstruction on the CUDA-enabled GPU. The proposed method is based on the FDK algorithm accelerated using two techniques: off-chip memory access reduction; and memory latency hiding. We have described how these techniques can be incorporated into CUDA code. The experimental results show that the proposed method takes 5.6 seconds to reconstruct a 512^3-voxel volume from 360 512^2-pixel projection images. This execution time is at least 18% faster than the prior methods [2,3], allowing us to obtain the entire volume immediately after a scan rotation of the flat panel detector. We also find that the filtering and backprojection performances are limited by the instruction issue rate and the memory bandwidth, respectively. With respect to acceleration techniques, memory access coalescing is essential to run the GPU as an accelerator for the CPU.

References

1. Kachelrieß, M., Knaup, M., Bockenbach, O.: Hyperfast parallel-beam and cone-beam backprojection using the cell general purpose hardware. Medical Physics 34(4), 1474–1486 (2007)
2. Xu, F., Mueller, K.: Real-time 3D computed tomographic reconstruction using commodity graphics hardware. Physics in Medicine and Biology 52(12), 3405–3419 (2007)
3. Scherl, H., Keck, B., Kowarschik, M., Hornegger, J.: Fast GPU-based CT reconstruction using the common unified device architecture (CUDA). In: Proc. Nuclear Science Symp. and Medical Imaging Conf (NSS/MIC 2007), October 2007, pp. 4464–4466 (2007)
4. Riabkov, D., Xue, X., Tubbs, D., Cheryauka, A.: Accelerated cone-beam backprojection using GPU-CPU hardware. In: Proc. 9th Int'l. Meeting Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine (Fully 3D 2007), July 2007, pp. 68–71 (2007)
5. Zhao, X., Bian, J., Sidky, E.Y., Cho, S., Zhang, P., Pan, X.: GPU-based 3D cone-beam CT image reconstruction: application to micro CT. In: Proc. Nuclear Science Symp. and Medical Imaging Conf. (NSS/MIC 2007), October 2007, pp. 3922–3925 (2007)
6. Schiwietz, T., Bose, S., Maltz, J., Westermann, R.: A fast and high-quality cone beam reconstruction pipeline using the GPU. In: Proc. SPIE Medical Imaging 2007, February 2007, pp. 1279–1290 (2007)
7. Gac, N., Mancini, S., Desvignes, M.: Hardware/software 2D-3D backprojection on a SoPC platform. In: Proc. 21st ACM Symp. Applied Computing (SAC 2006), pp. 222–228 (April 2006)
8. nVIDIA Corporation: CUDA Programming Guide Version 1.1 (November 2007), `http://developer.nvidia.com/cuda/`
9. Feldkamp, L.A., Davis, L.C., Kress, J.W.: Practical cone-beam algorithm. J. Optical Society of America 1(6), 612–619 (1984)
10. Li, M., Yang, H., Koizumi, K., Kudo, H.: Fast cone-beam CT reconstruction using CUDA architecture. Medical Imaging Technology 25(4), 243–250 (2007) (in Japanese)
11. Ikeda, T., Ino, F., Hagihara, K.: A code motion technique for accelerating general-purpose computation on the GPU. In: Proc. 20th IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS 2006), 10 pages (April 2006) (CD-ROM)
12. Grass, M., Köhler, T., Proksa, R.: 3D cone-beam CT reconstruction for circular trajectories. Physics in Medicine and Biology 45(2), 329–347 (2000)
13. Shepp, L.A., Logan, B.F.: The fourier reconstruction of a head section. IEEE Trans. Nuclear Science 21(3), 21–43 (1974)

Improving the Performance of Tensor Matrix Vector Multiplication in Cumulative Reaction Probability Based Quantum Chemistry Codes

Dinesh Kaushik[1], William Gropp[2], Michael Minkoff[1], and Barry Smith[1]

[1] Argonne National Laboratory, Argonne, IL 60439, USA
{kaushik,minkoff,bsmith}@mcs.anl.gov
[2] University of Illinois Urbana-Champaign, Urbana, IL 61801, USA
wgropp@uiuc.edu

Abstract. Cumulative reaction probability (CRP) calculations provide a viable computational approach to estimate reaction rate coefficients. However, in order to give meaningful results these calculations should be done in many dimensions (ten to fifteen). This makes CRP codes memory intensive. For this reason, these codes use iterative methods to solve the linear systems, where a good fraction of the execution time is spent on matrix-vector multiplication. In this paper, we discuss the tensor product form of applying the system operator on a vector. This approach shows much better performance and provides huge savings in memory as compared to the explicit sparse representation of the system matrix.

1 Introduction and Motivation

The prevalence of parallel processors makes many areas of simulation accessible that was only possible in the recent past on specialized facilities. One area of application is the use of computational methods to calculate reaction rate coefficients. These coefficients are often estimated experimentally. However, the simulations approaches [1,2] provide a reasonable alternative. Typically the ab initio approach is only applicable to small atomic systems. In these models the dimensionality of the problem is the number of degrees of freedom in the molecular system. If we consider torsion, stretching, etc., the maximum number of degrees of freedom (DOF) for a molecule is proportional to N, the number of atoms. Thus dealing with problems of only three to five degrees of freedom is quite restrictive. The alternative to ab initio methods is the use of statistical studies of reaction paths and thus obtain the reaction rate coefficients statistically. This approach is founded however on a less solid theoretical basis.

Reaction rate calculation involves a dimensional effect based upon DOF. That is we consider the reactions that involve molecules having various independent coordinates. For a simple two atom molecule in which we only consider one dimension and a variable representing the distance between the two atoms, we would have one DOF. However, if we add the angle between the atoms in two

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 120–130, 2008.

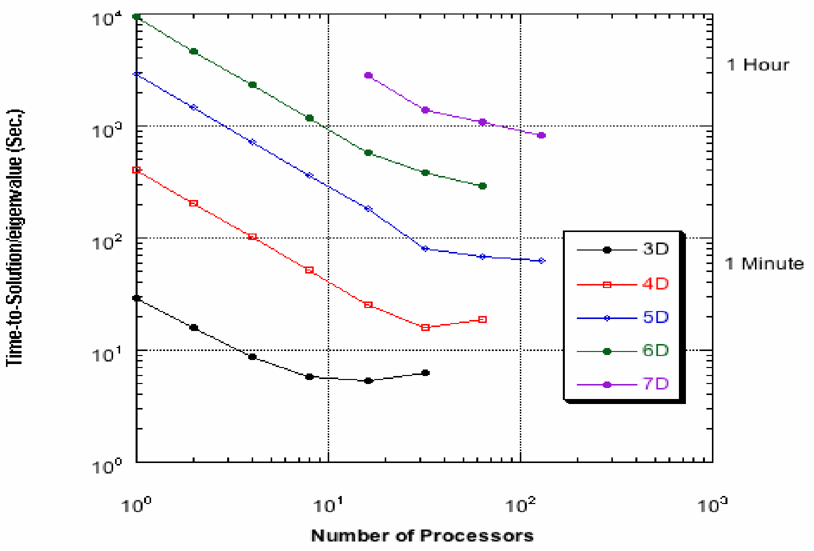


Fig. 1. Sample parallel performance of the CRP code on up to 128 processor of IBM SP3 at NERSC

dimensional space and also add the torsion effect we would have three DOF. We are interested in problems of up to ten or more DOF. This leads to large-scale problems in which parallel computation is a central aspect of the algorithmic approach. For such problems the Green's function solutions (see Section 2) cannot be done by direct linear solvers. A standard approach even applied to lower DOF is to use iterative methods such as GMRES [3] for solving the linear systems. The solution of these linear systems is the fundamental computational cost in the method as we and others have observed. In some of our computational experiments (see Figure 1) we have obtained an accurate eigenvalue in only two to three iterations, however we require from five hundred to a thousand GMRES iterations for each of the Green's function solves. Thus the principal focus of this paper is on studying an efficient implementation of matrix-vector multiplication.

Normally the matrix vector multiplication is done by first building up the large sparse matrix from the tensor products of one dimensional operators with the identity matrix. The sparse matrix vector product is well known to give poor performance since it is memory bandwidth limited computation with poor data reuse [4,5]. Since this kernel is responsible for a large fraction (over 80 %) of overall execution time, addressing its performance issues is crucial to obtain a reasonable percentage of machine peak. In this paper, we suggest an alternative approach (in Section 4) that transforms the memory bandwidth limited sparse matrix vector products to matrix-matrix multiplications with high level of data locality. This approach holds the potential to improve the performance of the overall code by a large factor.

The rest of the paper is organized as follows. We discuss the background of the CRP approach in Section 2.1. Next we analyze the performance characteristics of the sparse matrix vector multiplication approach in Section 3. We present the details of the tensor matrix vector multiplication approach in Section 4. Then we compare the performance of these two choices for matrix vector multiplication on Intel Madison processor in Section 5.

2 Background of the CRP Approach

The Cumulative Reaction Probability function is:

$$k(T) = [2\pi\hbar Q_r(T)]^{-1} \int_{-\infty}^{\infty} dEe^{-E/kT} N(E) \tag{1}$$

where Q_r is the reactant partition function. The rate constant is given as

$$k(E) = [2\pi\hbar\rho_r(E)]^{-1} N(E) \tag{2}$$

Therefore the CRP is key in calculating the rate constant. In fact, $N(E)$ can be expressed in terms of the trace of the reaction probability operator, $\hat{P}$

$$N(E) = tr[\hat{P}(E)] \equiv \sum kp_k(E) \tag{3}$$

and

$$\hat{P}(E) = 4\hat{\epsilon}_r^{1/2}\hat{G}(E)\hat{\epsilon}_p\hat{G}(E)\hat{\epsilon}_r^{1/2} \tag{4}$$

The Green's function is

$$\hat{G}(E) = (E + i\hat{\epsilon} - \hat{H})^{-1} \tag{5}$$

$\hat{H}$ is the Hamiltonian and $\hat{\epsilon} = \hat{\epsilon}_r + \hat{\epsilon}_p$ where $\hat{\epsilon}$ is a given absorbing potential, and $\hat{\epsilon}_r$ and $\hat{\epsilon}_p$ are, respectively are the part of $\hat{\epsilon}$ in the reactant and product regions (see [1,2,6] for details).

In summary, we seek to obtain the major components of the trace of $\hat{P}(E)$. Thus we seek the largest few eigenvalues of this operator. This can be accomplished by means of a Lanczos iteration of (4). For each Lanczos iteration we require the solution of two linear systems (5):

$$(E + i\hat{\epsilon} - \hat{H})\mathbf{y} = \mathbf{x} \tag{6}$$

and its adjoint when $\mathbf{x}$ is known. The matrix on the left hand side of Equation 6 is obtained from one dimensional operators as described next.

2.1 Matrix Vector Multiplication in CRP

For simplicity, let us consider a three dimensional system with n mesh points in each dimension. Then, we need to multiply matrix A $(n^3 \times n^3)$ with a vector $\mathbf{v}$ of size n^3.

$$\mathbf{w} = A\mathbf{v} \tag{7}$$

with $\mathbf{w}$ being the output vector of size n^3. The system matrix A is sparse with the following components:

$$A = B_z \otimes I \otimes I + I \otimes B_y \otimes I + I \otimes I \otimes B_x \tag{8}$$

Where, $\otimes$ denotes the tensor (Kronecker) product of one dimensional operators (B_x, B_y, B_z) with the identity matrix(I). The operators B_x, B_y, and B_z are dense matrices of size $n \times n$.

For d dimensions, we will have d terms in Equation 8 involving d tensor products of dense matrices of size $n \times n$ with the identity matrices of order n. As stated earlier, doing the matrix vector multiplication (Equation 7) is a key operation in the CRP algorithm. Next we discuss the sparse representation of matrix A.

3 Sparse Matrix Vector Product

The sparse matrix-vector product is an important part of many iterative solvers used in scientific computing. While a detailed performance modeling of this operation can be complex, particularly when data reference patterns are included [5,7,8], a simplified analysis can still yield upper bounds on the achievable performance of this operation. To illustrate the effect of memory system performance, we consider a generalized sparse matrix-vector multiply that multiplies a matrix by N vectors. This code, along with operation counts, is shown in Figure 2.

3.1 Estimating the Memory Bandwidth Bound

To estimate the memory bandwidth required by this code, we make some simplifying assumptions. We assume that there are no conflict misses, meaning that

```
for (i = 0, i < m; i++) {
  jrow = ia(i+1)                          // 1 Of, AT, Ld
  ncol = ia(i+1) - ia(i)                  // 1 Iop
  Initialize, sum1, ..., sumN             // N Ld
  for (j = 0; j < ncol; j++) {            // 1 Ld
    fetch ja(jrow), a(jrow),
          x1(ja(jrow)), ..., xN(ja(jrow)) // 1 Of, N+2 AT, N+2 Ld
    do N fmadd (floating multiply add)    // 2N Fop
    jrow++
  }                                       // 1 Iop, 1 Br
  Store sum1, ..., sumN in
        y1(i), ..., yN(i)                 // 1 Of, N AT, N St
}                                         // 1 Iop, 1 Br
```

Fig. 2. General form of sparse matrix-vector product algorithm: storage format is AIJ or compressed row storage; the matrix has `m` rows and `nz` non-zero elements and gets multiplied with `N` vectors; the comments at the end of each line show the assembly level instructions the current statement generates, where `AT` is address translation, `Br` is branch, `Iop` is integer operation, `Fop` is floating-point operation, `Of` is offset calculation, `LD` is load, and `St` is store

each matrix and vector element is loaded into cache only once. We also assume that the processor never waits on a memory reference, that is, any number of loads and stores can be issued in a single cycle.

For the algorithm presented in Figure 2, the matrix is stored in compressed row storage format (similar to PETSc's AIJ format [9]). For each iteration of the inner loop in Figure 2, we transfer one integer (`ja` array) and $N+1$ doubles (one matrix element and N vector elements), and we do N floating-point multiply-add (`fmadd`) operations or $2N$ flops. Finally, we store the N output vector elements. If we just consider the inner loop and further assume that vectors are in cache (and not loaded from memory), we load one double and one integer for $2N$ flops or 6 bytes/flop for one vector and 1.5 bytes/flop for four vectors (see [4] for more detailed treatment). The STREAM [10] benchmark bandwidth on Intel Madison processor is about 4,125 MB/s. This gives us the maximum achievable performance of 687 Mflops/s for one vector and 2,750 Mflops/s for four vectors while the corresponding observed numbers are 627 Mflops/s and 1,315 Mflops/s (out of the machine peak of 6 Gflops/s).

Following a similar procedure, we show the memory bandwidth bound, the actual performance and the peak performance for IBM Power 4, Intel Xeon, IBM BlueGene, and Intel Madison processors (assuming only one vector) in Figure 3. It is clear that the performance of sparse matrix vector multiplication is memory

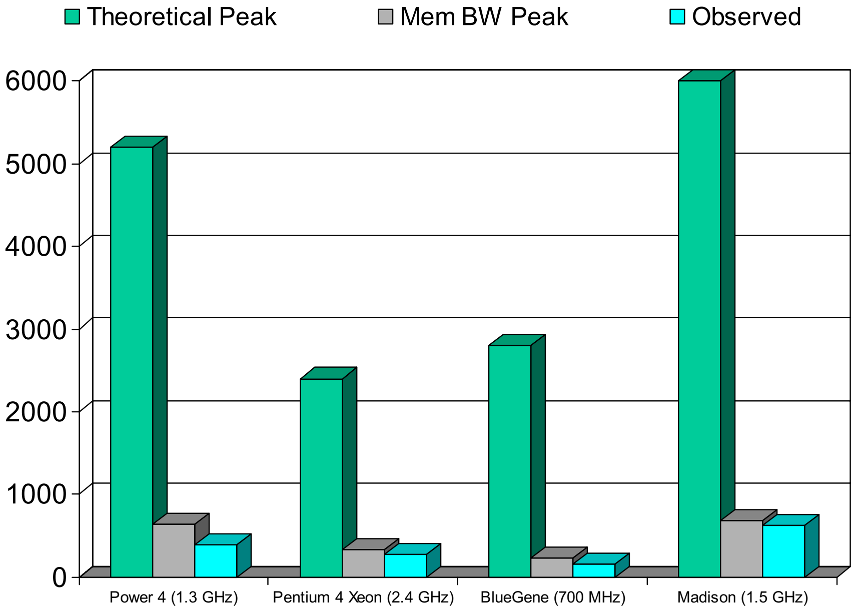


Fig. 3. Memory bandwidth bound for sparse matrix-vector product. Only one vector ($N = 1$) is considered here. The matrix size has `m` = 90,708 rows and `nz` = 5,047,120 nonzero entries. The processors are 1.3 GHz IBM Power 4, 2.4 GHz Intel Xeon, 700 MHz IBM BlueGene, and 1.5 GHz Intel Madison. The memory bandwidth values are measured using the STREAM benchmark.

bandwidth limited and the peak processor performance is pretty much irrelevant for this computation. We next discuss the tensor product form of the system operator that does not suffer from this limitation.

4 Tensor Matrix Vector Product

The system matrix in the CRP code comes from the tensor products of one directional dense operators with the identity matrix. This allows us to do the matrix vector multiplication without ever forming the large sparse matrix. Though a cheap approximation to the system matrix is usually needed for preconditioning purpose, we assume that it can be obtained in some suitable way (for example, see [11]) or one can possibly apply the same technique (of tensor matrix vector multiplication) while carrying out the matrix vector products of the preconditioned system.

We can combine the identity matrix tensor products in Equation 8 (and its higher dimensional counterparts). In general, the matrix vector product of Equation 7 will be the sum of the terms made from the three types of operations: $(I \otimes B)\mathbf{v}$, $(B \otimes I)\mathbf{v}$, and $(I \otimes B \otimes I)\mathbf{v}$. We describe how to carry out each of these operations efficiently next. The three dimensional case is described in detail in [12].

Type A: $(\boldsymbol{I} \otimes \boldsymbol{B})\mathbf{v}$

We need to evaluate

$$(I_{p\times p} \otimes B_{m\times m})\mathbf{v}$$

with $\mathbf{v} = (v_1, v_2, ..., v_{pm})^T$. We can view the vector $\mathbf{v}$ as a matrix (V) of size $m \times p$ and then

$$(I_{p\times p} \otimes B_{m\times m})\mathbf{v} = B_{m\times m} \times V_{m\times p}$$

It should be noted that the memory layout of the vectors $\mathbf{v}$ and $\mathbf{w}$ does not change in this operation. Since the matrix V is stored columnwise, its data access pattern in the above matrix-matrix multiplication is ideal (unit stride). As the number of dimensions increases, the order (p) of the identity matrix gets larger and larger. Therefore, the above algorithm multiplies a small square matrix (B) with a highly rectangular matrix (V) for large dimensions. We will see in Section 5 that many matrix-matrix multiplication implementations do not perform well under this situation.

Type B: $(\boldsymbol{B} \otimes \boldsymbol{I})\mathbf{v}$

Here we need to evaluate

$$(B_{m\times m} \otimes I_{p\times p})\mathbf{v}$$

We can view the vector $\mathbf{v}$ as a matrix (V) of size $p \times m$ and then

$$(B_{m\times m} \otimes I_{p\times p})\mathbf{v} = V_{p\times m} \times {B^T}_{m\times m}$$

where ${B^T}_{m\times m}$ is the transpose of $B_{m\times m}$ ([12]). Again, the memory layout of the vector **v** and **w** does not change with this operation and this is also a matrix-matrix multiplication. The data access pattern for the matrix V is not unit stride here (with the normal triply nested loop implementation) and transposing this matrix may bring significant performance gains.

Type C: $(\boldsymbol{I} \otimes \boldsymbol{B} \otimes \boldsymbol{I})\mathbf{v}$

Here we need to evaluate

$$(I_{p\times p} \otimes B_{m\times m} \otimes I_{r\times r})\mathbf{v}$$

with $\mathbf{v} = (v_1, v_2, ..., v_{pmr})^T$.

This can be evaluated by looping over Type B term algorithm p times [12]. Each iteration of this loop will evaluate the Type B term $V_{r\times m} \times {B^T}_{m\times m}$. Again this can be done without changing the memory layout of the vectors **v** and **w**.

5 Results and Discussion

In the previous section, we saw that all terms of the generalized form (for d dimensions) of Equation 8 can be evaluated as dense matrix-matrix multiplication,

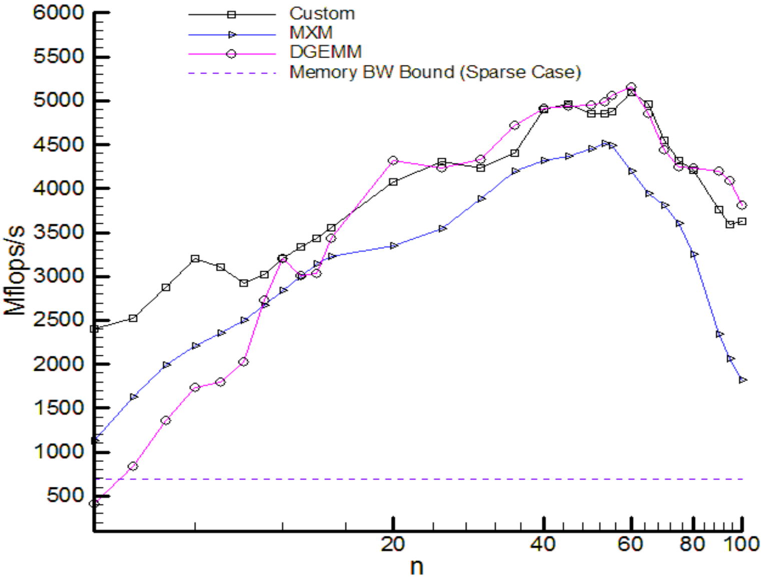


Fig. 4. Performance of the tensor matrix vector multiplication for three dimensions on Intel Madison (1.5 GHz) processor. The custom code is manually optimized code, MXM code is from [12] and DGEMM() routine is from Intel's MKL library. Note that the sparse matrix vector multiplication will only do at most about 687 Mflops/s based on the memory bandwidth bound on this processor.

which inherently has very high data reuse and usually performs at a large fraction of machine peak (if implemented properly). We present here some sample performance results on Intel Madison processor (1.5 GHz, 4 MB L2 cache, and 4GB memory). We discuss three implementations:

- **Custom code**: this is the hand optimized code specifically written for evaluating the Type A, B, and C terms.
- **MXM code**: this is taken from Deville, et al. [12].
- **DGEMM**: this is from a vendor library (Intel MKL).

We show the performance advantage of the tensor matrix vector multiplication in three dimensions for n = 5 to 100 in Figure 4. If we had done the matrix vector multiplication by explicitly building the sparse matrix, the performance would have been limited to about 687 Mflops/s (see the dotted line in Figure 4, which is based on the memory bandwidth bound) on this processor. All the three variants give good performance for reasonably large n (≥ 15). Note that there are slightly more floating point operations while doing the tensor matrix vector multiplication as compared to the explicit sparse matrix formation case. However, the execution time is less for the former since the computation is cpu bound and not memory bandwidth limited (which is the case for the later).

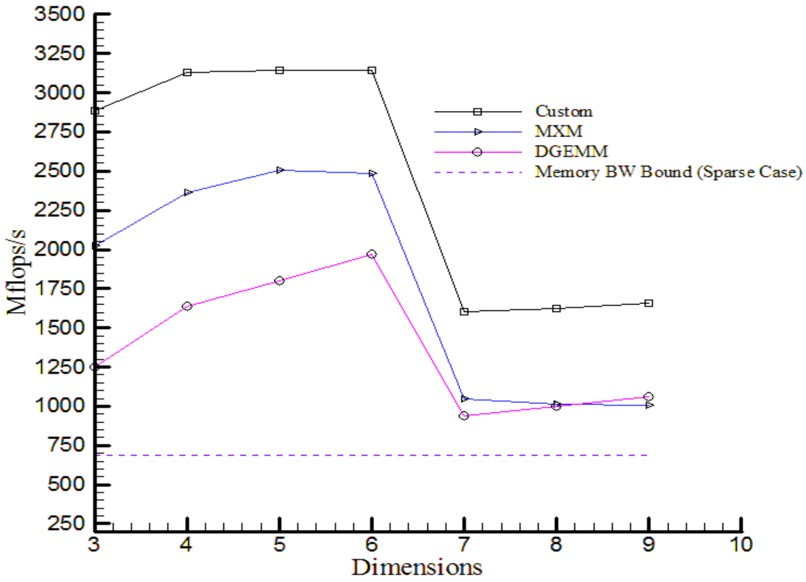


Fig. 5. Performance of the tensor matrix vector multiplication for n = 7 in all dimensions. The sharp drop in performance is due to the working set of the problem going out of the L2 cache (4 MB) of the Intel Madison processor. We are trying to contain this drop (to some extent) with better implementation (with extra blocking). Notice that the DGEMM() does not perform well for small values of matrix sizes and especially when the two matrix sizes are vastly different (large dimension case).

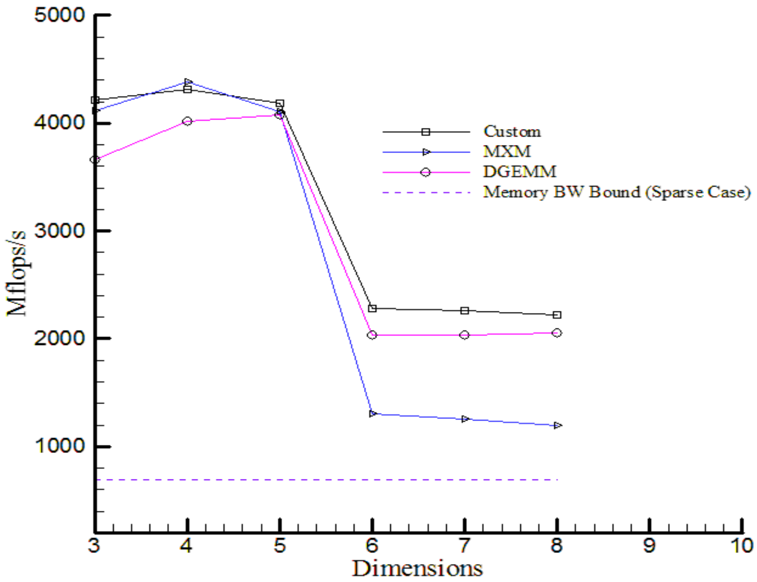


Fig. 6. This case has 51 points along the reaction path and 7 points in other dimensions. This represents the CRP code more closely. The performance advantage of the tensor matrix vector multiplication over the sparse approach is still maintained.

While vendors have invested considerable effort in optimizing the matrix-matrix multiplication, it is usually done for large and balanced matrix sizes. The CRP code involves matrix-matrix multiplications between small square matrices (typically 7×7 to 10×10) and highly rectangular matrices (arising from the matrix view of the input vector $\mathbf{v}$). We show this situation in Figure 5 for $n = 7$. The DGEMM gives the worst performance of all for this case, especially for higher dimensions (when the matrix coming from the input vector becomes very elongated, e.g., 7×7^7 for eight dimensional problem). The custom code also shows sharp drop in performance (typically characteristic of the working set getting out of a fast memory level). We are trying some other implementations to reduce this performance drop.

Figure 6 shows the same scenario as in Figure 5 except that there are more mesh points (51) along the reaction coordinate than in the other directions (7). This is more consistent with the linear systems being solved in the CRP code (Figure 1). Again the performance is much better with the custom code than is possible with the corresponding sparse matrix-vector multiplication code (the dotted line in Figure 6).

5.1 Storage Advantage

The chemistry codes work with many dimensions and are memory intensive for that reason. If we never form the large sparse system matrix, there is huge saving in memory. The memory needed for tensor representation of the operator in d dimensions is $O(dn^2)$ while it will be $O(n^{d+1})$ if we explicitly store it as sparse matrix. Therefore, the tensor product form of the operator will allow larger problems to be solved for the same amount of available memory.

6 Conclusions and Future Work

We have demonstrated memory and performance advantages of applying the system operator in the tensor product form (rather than as a sparse matrix). Since matrix-vector multiplication takes a large chunk of the overall execution time, a big improvement in the overall performance of the CRP code is expected when the tensor product form of the operator is employed. Further, this technique can be applied to any discretization scheme where the system matrix originates from some form of tensor products of smaller dense matrices (and work is in progress to demonstrate its applicability in a real application code). This paper has compared the performance of some implementations of matrix-matrix product for small size matrices. We observe that many common implementations of this operation do not perform well for small size and highly rectangular matrices. In future, we will evaluate some more competing implementations such as transposing the input vector for a more efficient evaluation of Type B terms, doing more blocking to contain the performance drops when the computation goes out of L2 cache, and DGEMM from some other libraries.

Acknowledgments

We thank Paul Fischer, Ron Shepard, and Al Wagner of Argonne National Laboratory for many helpful discussions. The computer time was supplied by DOE (through Argonne, NERSC, and ORNL) and NSF (through Teragrid at SDSC).

References

1. Wyatt, R.E., Zhang, J.Z.H.: Dynamics of molecules and chemical reactions. CRC Press, Boca Raton (1996)
2. Manthe, U., Miller, W.H.: The cumulative reactions probability as eigenvalue problem. J. Chem. Phys., pp. 3411–3419 (1999)
3. Saad, Y., Schultz, M.H.: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM Journal of Scientific and Statistical Computing 7(3), 856–869 (1986)
4. Gropp, W.D., Kaushik, D.K., Keyes, D.E., Smith, B.F.: Toward realistic performance bounds for implicit CFD codes. In: Keyes, D., Ecer, A., Periaux, J., Satofuka, N., Fox, P. (eds.) Proceedings of Parallel CFD 1999, pp. 233–240. Elsevier, Amsterdam (1999)
5. Toledo, S.: Improving the memory-system performance of sparse-matrix vector multiplication. IBM Journal of Research and Development 41, 711–725 (1997)
6. Miller, W.H.: Quantum and semiclassical greens functions in chemical reaction dynamics. J. Chem. Soc., Faraday Trans. 93(5), 685–690 (1997)

7. Temam, O., Jalby, W.: Characterizing the behavior of sparse algorithms on caches. In: Proceedings of Supercomputing 1992, pp. 578–587. IEEE Computer Society, Los Alamitos (1992)
8. White, J., Sadayappan, P.: On improving the performance of sparse matrix-vector multiplication. In: Proceedings of the 4th International Conference on High Performance Computing (HiPC 1997), pp. 578–587. IEEE Computer Society, Los Alamitos (1997)
9. Balay, S., Buschelman, K.R., Gropp, W.D., Kaushik, D.K., Knepley, M.G., McInnes, L.C., Smith, B.F.: PETSc home page (2002), `http://www.mcs.anl.gov/petsc`
10. McCalpin, J.D.: STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia (1995), `http://www.cs.virginia.edu/stream`
11. Poirier, B.: Efficient preconditioning scheme for block partitioned matrices with structured sparsity. Numerical Linear Algebra with Applications 7, 1–13 (2000)
12. Deville, M.O., Fischer, P.F., Mund, E.H.: High-Order Methods for Incompressible Fluid Flow. Cambridge University Press, Cambridge (2002)

Experimental Evaluation of Molecular Dynamics Simulations on Multi-core Systems

Sadaf R. Alam, Pratul K. Agarwal, Scott S. Hampton, and Hong Ong

Computer Science and Mathematics Division,
Oak Ridge National Laboratory, Oak Ridge, USA
{alamsr,garwalpk,hamptonss,hongong}@ornl.gov

Abstract. Multi-core processors introduce many challenges both at the system and application levels that need to be addressed in order to attain the best performance. In this paper, we study the impact of the multi-core technologies in the context of two scalable, production-level molecular dynamics simulation frameworks. Experimental analysis and observations in this paper provide for a better understanding of the interactions between the application and the underlying system features such as memory bandwidth, architectural optimization, and communication library implementation. In particular, we observe that parallel efficiencies could be as low as 50% on quad-core systems while a set of dual-core processors connected with a high speed interconnect can easily outperform the same number of cores on a socket or in a package. This indicates that certain modifications to the software stack and application implementations are necessary in order to fully exploit the performance of multi-core based systems.

Keywords: Multicore, Performance, Molecular Dynamics Simulation, HPC.

1 Introduction

The shift in processor architecture from the traditional improvement in clock speed to using multiple cores provides a solution to increase the performance capability on a single chip without introducing a complex system and increasing the power requirements. As a result, multi-core processors have emerged as the dominant architectural paradigm for both desktop and high-performance systems. However, multi-core systems have presented many challenges in maximizing application performance. Therefore, it is important to identify the factors that could potentially limit the performance and scalability of application through benchmarking.

Most existing benchmarks are not targeted specifically at multi-core architectures and thus are not able to expose potential limitations of these devices such as shared cache coherence overhead, memory resource contention, and intra- as well as inter-communication bottlenecks. Consequently, a systematic evaluation of these processors using pertinent applications to discover the critical performance path is crucial to discovering problem areas. In this study, we present a methodology for characterizing the performance of a diverse range of multi-core devices in the context of two scalable, production-level molecular dynamics (MD) simulation frameworks: AMBER

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 131–141, 2008.

PMEMD [1] and LAMMPS [2]. Our aim is to improve the overall MD simulation performance on multi-core systems through finding answers to the following questions:

- Can the MD simulation scale well on a multi-core system?
- What are the factors that affect the MD simulation performance/scaling on a multi-core system?
- Are these factors due to the implementation of the MD simulation, the underlying system services, or both?
- How could these limiting factors be avoided?

The rest of this paper is organized as follows: We briefly introduce the background knowledge of MD simulation in Section 2. The experimental methodology is explained in Section 3. The evaluation, results, and analysis are presented in Section 4 and Section 5. We discuss our observations and analysis in Section 6. Finally, we conclude and indicate future work directions in Section 7.

2 Background

MD is being widely used to study everything from material properties, to protein folding, and even drug design. Briefly, it is a method for studying the properties of a set of particles as they evolve over time. The particles interact through various pair-wise forces according to Newton's second law of motion, which states that the sum of the forces on an object is equal to the product of its mass and acceleration, i.e., $F = ma$.

During an MD run, the positions and velocities of the system are used to compute instantaneous averages of macroscopic properties such as potential and kinetic energies, pressure, and temperature. From a computational perspective, there are two key challenges for MD programs. First, it is important to compute the forces with a relatively high degree of accuracy. Unfortunately, this is an expensive operation, as each atom needs to know the position of the other N-1 atoms. In the worst case, calculating the forces is an $O(N^2)$ algorithm. However, there exist efficient approximations that can lower the algorithmic cost to O(N) for large N.

The second challenge is that MD often requires a large amount of computational time in order to accurately measure desired quantities. For example, in biological simulations, the typical MD step simulates about 10^{-15} seconds of real time. Events of biological interest typically occur on the scale of 10^{-9}–10^{-3} seconds or longer. This difference of 6 to 12 orders of magnitude suggests that MD would not be possible if it was not for parallel computing [3][4].

3 Experimental Methodology

In this study, we attempt to understand the performance of two scalable, production-level MD simulations on multi-core systems. Our target platforms include a four-socket, dual-core AMD Opteron 8216 system [5], a quad-core Intel Clovertown processor system [6], and dual-core Opteron Cray XT3 and XT4 systems [7][8]. Cray XT4 contains Rev F Opteron while the Cray XT3 system is composed of an earlier release of dual-core Opteron processors. The system configuration details are listed in Table 1.

Table 1. Target systems

	AMD Opteron 8216	Intel Clovertown	Cray XT3 (Dual-core Opteron)	Cray XT4 (Dual-core Opteron)
Clock frequency	2.6 GHz	2.4 GHz	2.6 GHz	2.6 GHz
L1 Cache	128 Kbyte	16 Kbyte	64 Kbytes	64 Kbytes
L2 Cache	1 Mbyte	4 Mbytes per die, 8 Mbytes total	1 Mbytes	1 Mbytes
Memory bandwidth	10.6 GB/s	17 GB/s (FSB) at 1066 MHz	6.4 GB/s	10.6 GB/s
Operating System	Linux	Linux	Catamount	Catamount
Compiler	PGI	Intel	PGI	PGI
MPI library	MPICH2	Intel MPI	Cray MPI	Cray MPI

From the hardware perspective, there are key differences between Intel and AMD processor designs that can influence the application performance. For instance, the AMD systems are considered as NUMA (Non Uniform Memory Architecture) because it provides a memory controller as part of the direct connect architecture, which gives each core in a multi-core processor low latency and high-bandwidth connections to its local memory stores. The other approach, adopted by Intel in its Xeon and Pentium processors, creates a single shared pool of memory, which all the cores access memory through an external front-side bus (FSB) and a memory controller hub.

From the software stack perspective, there are also some notable differences. The Cray XT systems use a lightweight kernel called Catamount, while the other two systems run standard Linux distribution. The systems have different Math library: the Cray XT systems have the AMD‘s ACML API, the Clovertown system has the Intel’s MKL, and the Opteron system has the ATLAS package. The systems also have different MPI library: the Opteron system has the MPICH library compiled with the PGI compiler, the Clovertown system has the Intel MPI library, and the Cray XT systems have an MPI-2 compliant library, which is derived from the MPICH-2 library. Furthermore, the Cray XT systems allow additional control over MPI task placement schemes.

For measurements, we first ascertain system characteristics using a set of micro benchmarks from the HPC Challenge benchmark suite (HPCC) [9], cache benchmark, and MPI performance benchmarks [10]. We anticipate that the memory bandwidth on the multi-core systems is one of the major issues. In particular, we expect that the memory bandwidth per core would not scale with the floating-point performance per socket. However, it is unclear how this will impact different memory access patterns within an application. We therefore use the HPCC benchmark suite to characterize application kernels in terms of spatial and temporal locality of their memory access patterns. For instance, DGEMM and High Performance Linpack (HPL) exhibit high spatial and temporal locality. The Fast Fourier Transform (FFT) calculations show high temporal but poor spatial locality. Lastly, we want to understand how the MD application performance and scaling achieved on these systems are sensitive to some key features of the target multi-core devices.

4 Experimental Results and Analysis

4.1 Notation and Test Parameters

The HPCC benchmark suite tests multiple system performance attributes and provides performance indicators for processing power, interconnect, and memory subsystems. In the context of multi-core system performance, experiments can be configured to run in serial (one core per socket), embarrassingly parallel (work per core remains same and no interactions between cores), and strong scaling (all cores contribute to a given task).

The naming convention we used for serial runs is '*_single*', embarrassingly parallel is '*_Star*', and strong scaling mode is '*_MPI*'. For the Cray XT results, we choose two placement schemes namely '*_default*' and '*_reorder*'. The default scheme allocates MPI tasks in a round-robin scheme, for example, MPI task 0 is mapped on core 0 of processor 0 and MPI task 1 will be mapped to core 0 of processor 1. In the reorder scheme MPI tasks 0 and 1 are assigned to core 0 and 1 of processor 0. In total, 8 XT3 and XT4 cores are used.

4.2 Cache Characteristics

We ran *cachebench* in serial (see left graph of Figure 1) and Embarrassingly Parallel (EP) or Star mode (see right graph of Figure 1) on our target systems. On the Clovertown system we observe a very high bandwidth on up to 32K vector lengths accesses but then drops significantly as the L2 cache is saturated. Poor performance scaling for very large vectors and the difference in single and EP modes performance is only noted for very large vectors.

4.3 Memory Characteristics

HPL benchmark captures the peak floating-point performance of a system by solving a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers.

Figure 2 shows HPL performance in GFLOPS/s. These results are gathered on 16 XT cores, and 8 Opteron 8216 and Clovertown cores. The size of the matrix is an input parameter, which is slightly smaller for the 8 cores runs as compared to the 16 cores runs. There is approximately 2% difference between XT4 and XT3 runtimes and the placement scheme has negligible impact on the HPL performance. Opteron performance is approximately 3% lower than the Clovertown performance.

Figure 3 shows DGEMM performance in GFLOPS per core for serial and embarrassingly parallel (EP) execution modes. DGEMM (double-precision general matrix multiply) is a test similar to the HPL benchmark in that it tests the system for high local and spatial memory locality calculations. The results show that the Clovertown system attains maximum DGEMM performance as compared to the other systems. On the other hand, it shows nearly 10% performance variation in the EP execution mode, while the difference on the Opteron based systems was negligible. This result clearly shows benefit and impact of a large, shared L2 cache in the Clovertown design.

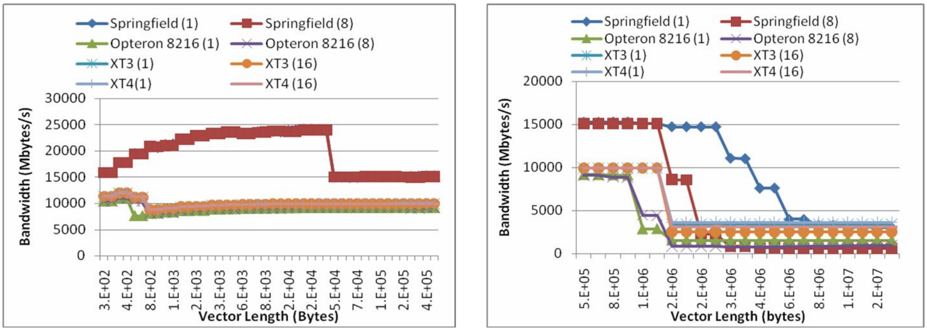


Fig. 1. Cache bench performance (Mbytes/s)

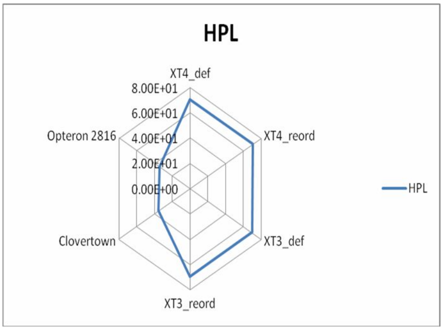


Fig. 2. HPL Performance (GFLOPS/s)

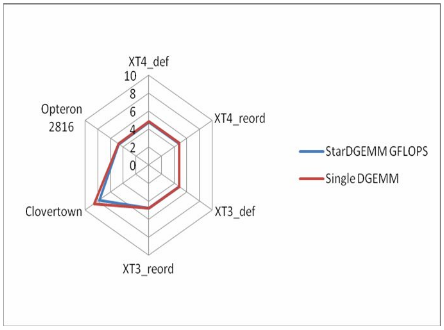


Fig. 3. DGEMM performance (GFLOPS/s)

The Fast Fourier Transform (FFT) benchmark has a high temporal but low spatial locality. Figure 4 show global, EP, and single-core view of the performance on 16 cores on the Cray XT systems and 8 cores on the standalone Opteron and Clovertown platforms. Clovertown shows the highest serial version performance and the lowest performance in global and EP modes. We also note that the placement of the MPI tasks result in about 10% performance variations on the Cray XT system runs.

4.4 Communication Characteristics

We then focused on communication performance, namely the MPI benchmarks. The HPC MPI test cases measure two different communication patterns. First, it measures the single-process-pair latency and bandwidth. Second, it measures the parallel all-processes-in-a-ring latency and bandwidth. For the first pattern, ping-pong communication is used on a pair of processes. In the second pattern, all processes are arranged in a ring topology and each process sends and receives a message from its left and its right neighbor in parallel. Two types of rings are reported: a naturally ordered ring (i.e., ordered by the process ranks in MPI_COMM_WORLD), and the geometric mean of the bandwidth of ten different randomly chosen process orderings in the ring.

Figure 5 shows that the Cray XT4 system benefits from the higher injection bandwidth of its SeaStar2 network. In fact, the SeaStar2 increases the peak network injection bandwidth of each node from 2.2 GB/s to 4 GB/s when compared to SeaStar, and increases the sustained network performance from 4 GB/s to 6 GB/s. Opteron performance is slightly higher than the Clovertown's performance except in the case of naturally ordered ring bandwidth. Overall these results show that the MPI communication performance is sensitive to the placement of the MPI tasks and the communication pattern. However, the difference in performance behavior is not apparent from the HPCC benchmark results. We therefore collected additional data using the Intel MPI benchmark (IMB) suite that reports latencies for different MPI communication patterns [4].

MPI applications typically make use of collective communication operations such as MPI_Allreduce. The latencies for MPI_Allreduce operations generated by the IMB benchmark are shown in Figure 6 for 16 cores on the Cray XT systems and 8 cores for the standalone multi-core systems. The Clovertown system has the lowest latencies for the MPI_Allreduce operations using Intel MPI library. Since the MPI latencies depend on a deep software and hardware stack, it cannot be attributed to a single hardware of software feature of the target systems.

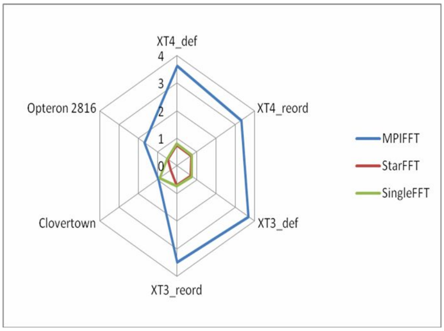


Fig. 4. FFT performance (GFLOPS/s)

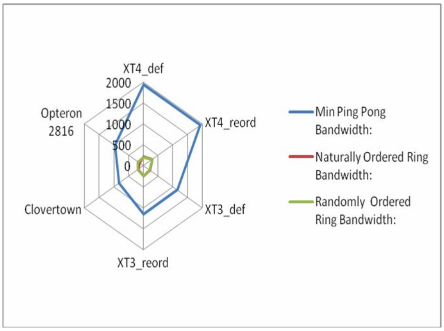


Fig. 5. MPI bandwidth (Mbytes/s)

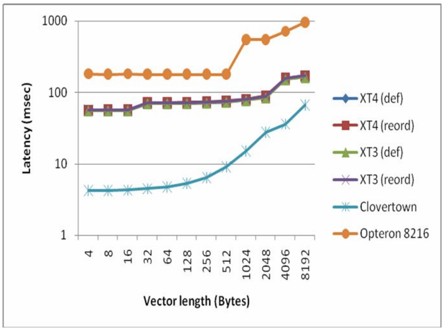


Fig. 6. MPI_Allreduce Latency (msec)

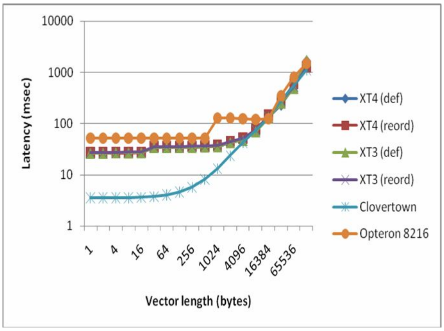


Fig. 7. IMB Exchange benchmark latency (msec)

Similarly, typical MPI applications perform data exchange between groups or pairs. The IMB exchange benchmark exhibits behavior of boundary exchange pattern in periodic manner. Figure 7 shows latencies of the MPI exchange benchmark. Like the MPI_Allreduce results, the Intel system outperforms the other systems for small message sizes. However, for large messages, there is very little variation on different target systems.

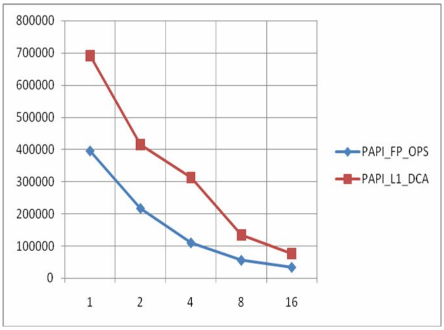


Fig. 8. Hardware counter values for LAMMPS per MPI task (JAC benchmark)

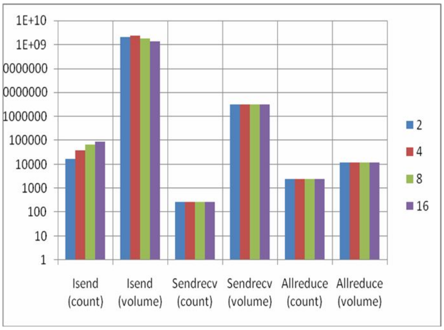


Fig. 9. MPI communication profile per MPI task for LAMMPS (JAC benchmark)

5 Application Performance and Scaling

We next characterize our two MD simulations, AMBER PMEMD (F90 and MPI) and LAMMPS (C++ and MPI), in terms of their computation, memory and communication requirements. For both simulations, we use an identical input deck, namely the Joint Amber Charmm (JAC), which is publicly available [11]. Furthermore, we profile these simulations using a set of performance tools that are composed of the hardware counter from PAPI [12] and the MPI profiling interface. The MPI requirements are independent of the target platform but the computation and memory access requirements depend on the compiler infrastructure. We normalize these requirements for the three systems.

Figure 8 shows floating-point requirements and data cache access requests for LAMMPS when running JAC benchmark for 1 psec simulation. Because of the inherent C++ pointer structure, the memory access operations are higher than the number of floating-point operations. We also note that both computation and memory operations requirements scale with the number of MPI tasks or cores.

Communication profile for the JAC experiment is shown in Figure 9. There are two dominant point-to-point communication patterns, non-blocking send followed by a blocking receive and Sendrecv and MPI_Allreduce collective communication operations in which all MPI tasks participate. The non-blocking send operations per MPI task increase with the 2^(MPI tasks), while the others operation count and sized remain constant.

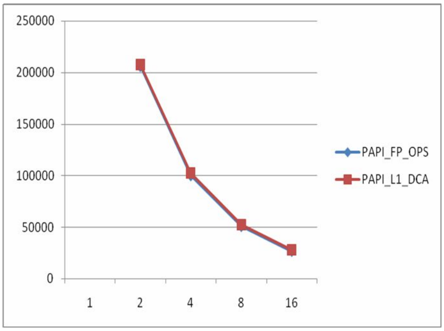


Fig. 10. Number of floating-point and memory access operations for PMEMD (JAC benchmark)

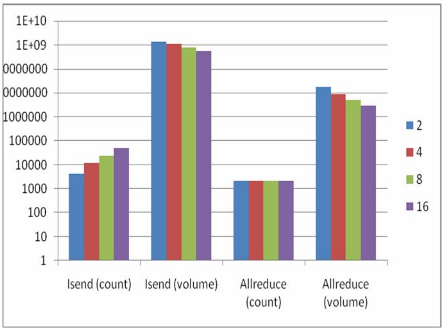


Fig. 11. Communication profile for PMEMD (JAC benchmark)

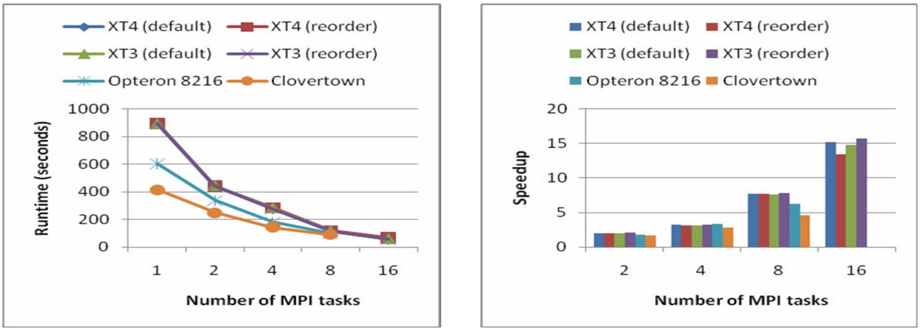


Fig. 12. Performance and speedup with compiler optimization (JAC with LAMMPS)

Similarly we obtain runtime profile for PMEMD experiments shown in Figure 10. Unlike LAMMPS, the floating-point computation and memory access requirements in the Fortran and MPI application are balanced and scale with the number of MPI tasks. Figure 11 shows communication profile for PMEMD experiments. Only two types of messages are dominant in the calculation: non-blocking point-to-point messages and collective MPI_Allreduce operations. Overall the messaging pattern is similar to LAMMPS but the behavior of collective communication operations is slightly different. Although the message sizes per processors decrease with the increase of the number of MPI tasks, the aggregate message count to volume ratio is higher than the LAMMPS simulation.

Figure 12 shows runtime and speedup for LAMMPS runs with optimization on. The single-core performance was optimal on the Clovertown platform but the speedup numbers are the lowest. On the Cray XT platforms, the runtimes are similar but slight variation is shown in the speedup. Overall the dual core performance is optimal on standalone systems but scaling is optimal on systems connected with a high speed interconnect.

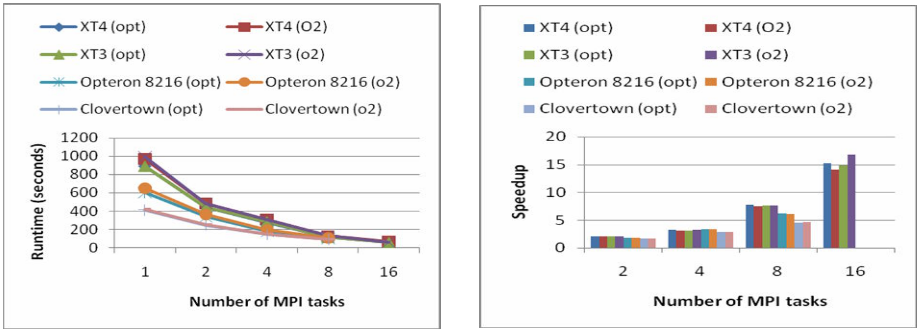


Fig. 13. Performance and scaling comparison with and without compiler optimization

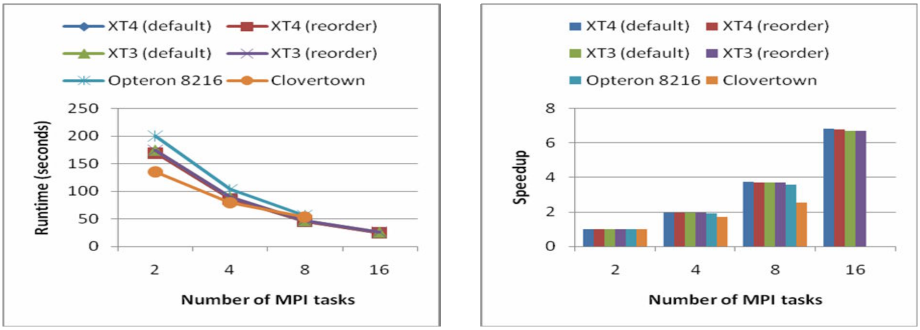


Fig. 14. Performance and scaling in the optimized mode running PMEMD (JAC benchmark)

Figure 13 compares performance and speedup on the optimized and un-optimized versions of the applications. Although the compiler optimization results are apparent on small number of cores, the shared memory access and communication costs offset the sophisticated compiler optimization.

Performance and scaling of the PMEMD test case are shown in Figure 14. The lower performance on the Opteron core in this case is attributed to the MPI_Allreduce latencies, which are significantly higher as the message sizes and the number of MPI tasks increase.

We attempted several placement schemes but there were no notable changes in MPI_Allreduce latencies. Additionally, the impact of a high bandwidth interconnect is shown in the Cray XT as the performance of 4 dual-core Opteron processors exceeds the 4 quad-core Clovertown processors. We also attempted both single and dual core processor placement on the Cray XT systems; however, there is negligible runtime difference when both cores and a single core participate in the calculations. Cray XT4 times are about 5% higher then the subsequent Cray XT3 times.

6 Discussion

The results with an identical test case using two scalable simulation frameworks on a diverse set of multi-core technologies reveal several key workload characteristics and impact of their mapping on the multi-core devices. Here we attempt to describe these findings in terms of the micro benchmark results that are collected on our target systems. Note that there are a numbers of parameters that are fixed for these experiments including compiler infrastructure and runtime systems.

First, we found by executing the HPCC memory locality benchmarks that the memory bandwidth on the applications is the lowest on the Intel Clovertown platform, especially when all cores participate in running the benchmark in embarrassingly parallel mode. This characteristic is shown in the scaling of both applications. Second, we note that the performance and scaling of the HPCC FFT benchmarks is representative of the MD applications performance. Another finding was the impact of a poor implementation of an MPI collective operation. PMEMD shows sensitivity to this metric while the LAMMPS code that only involves exchanging small size messages did not show an impact of this workload characteristic. We also demonstrate that high-level of compiler level optimization that are targeted towards better exploitation of the execution units only provide marginal benefits, particularly at the higher core count. Finally, we found that HPL and DGEMM based benchmarks are not representative of the MD workloads and cannot be considered for performance projections on emerging multi-core processors.

This study enables us the answer some of the questions that we posed earlier. We demonstrate that the scaling of the MD applications on multi-core systems, particularly on four and more cores require some reorganization of data structures and subsequently the access patterns to achieve high performance and parallel efficiencies. Furthermore, we note that the there are two key factors that limit the scaling on multi-core system: memory bandwidth and MPI collective operations performance. The high cache bandwidth as shown by a cache performance benchmark does not reflect the runtime characteristics of MD applications. MPI collective performance on the other hand does not impact both applications in a similar manner. Hence, the performance and scaling limiting factors are not only induced by system and software stack but also by the application implementation.

7 Conclusions

In this paper, we have chosen the HPC Challenge (HPCC) benchmark suite to test the system performance including floating point, compute power, memory subsystem, and global network issues. We have also chosen the JAC, which contains approximately 24K atoms, as an input to the two parallel versions of the MD simulation frameworks: Amber PMEMD and LAMMPS. From our experiments, we find that parallel efficiencies could be as low as 50% on the quad-core systems while a set of dual-core processors connected with a high speed interconnect can easily outperform the same number of cores on a socket or in a package. This trend indicates that optimizing intra-node communication is as important as optimizing inter-node communication.

For our future work, we plan to parameterize the runtime system to identify the impact of their performance attributes on the molecular dynamics applications. After identifying the performance and scaling limiting factors, we plan to propose and implement approaches for avoiding and reducing the performance limiting factors at the application and system software levels for a range of production-level MD simulation frameworks.

Acknowledgments

The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. The authors would like to thank National Center for Computational Sciences (NCCS) for access to Cray XT systems and support (INCITE award).

References

1. Pearlman, D.A., Case, D.A., Caldwell, J.W., Ross, W.S., Cheatham, T.E., Debolt, S., Ferguson, D., Seibel, G., Kollman, P.: AMBER, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules. Comput. Phys. Commun. 91, 1–41 (1995)
2. Plimpton, S.J.: Fast Parallel Algorithms for Short-Range Molecular Dynamics. J. Comp. Phys. 117, 1–19 (1995), `http://www.cs.sandia.gov/~sjplimp/lammps.html`
3. Agarwal, P.K., Alam, S.R.: Biomolecular simulations on petascale: promises and challenges. Journal of Physics: Conference Series (SciDAC 2006) 46, 327–333 (2006)
4. Alam, S.R., Agarwal, P.K., Giest, A., Vetter, J.S.: Performance Characterization of Biomolecular Simulations using Molecular Dynamics. In: ACM Symposium on Principle and Practices of Parallel Programming (PPOPP) (2006)
5. AMD Opteron Rev. F, `http://multicore.amd.com/`
6. Ramanthan, R.M.: Intel Multi-core Processors: Making the move to Quad-core and Beyond, white paper, `http://www.intel.com/technology/architecture/downloads/quad-core-06.pdf`
7. Vetter, J.S., et al.: Early Evaluation of the Cray XT3. In: IPDPS (2006)
8. Alam, S.R., Barrett, R.F., Kuehn, J.A., Roth, P.C., Vetter, J.S.: Characterization of Scientific Workloads on Systems with Multi-Core Processors. In: IEEE International Symposium on Workload Characterization (2006)
9. HPC Challenge benchmarks, `http://icl.cs.utk.edu/hpcc/`
10. Intel MPI Benchmarks, `http://www.intel.com/cd/software/products/`
11. MD benchmarks for AMBER, CHARMM and NAMD, `http://amber.scripps.edu/amber8.bench2.html`
12. Performance Application Programming Interface (PAPI), `http://icl.cs.utk.edu/papi/`

Parsing XML Using Parallel Traversal of Streaming Trees

Yinfei Pan, Ying Zhang, and Kenneth Chiu

Department of Computer Science, State University of New York,
Binghamton, NY 13902

Abstract. XML has been widely adopted across a wide spectrum of applications. Its parsing efficiency, however, remains a concern, and can be a bottleneck. With the current trend towards multicore CPUs, parallelization to improve performance is increasingly relevant. In many applications, the XML is streamed from the network, and thus the complete XML document is never in memory at any single moment in time. Parallel parsing of such a stream can be equated to parallel depth-first traversal of a streaming tree. Existing research on parallel tree traversal has assumed the entire tree was available in-memory, and thus cannot be directly applied. In this paper we investigate parallel, SAX-style parsing of XML via a parallel, depth-first traversal of the streaming document. We show good scalability up to about 6 cores on a Linux platform.

1 Introduction

XML has become the de facto standard for data transmission in situations requiring high degrees of interoperability, flexibility and extensibility, and exhibiting large degrees of heterogeneity. Its prevalence in web services has contributed to the success of large-scale, distributed systems, but some of the very characteristics of XML that have led to its widespread adoption, such as its textual nature and self-descriptiveness, have also led to performance concerns [5,6]. To address these concerns, a number of approaches have been investigated, such as more efficient encodings of XML to reduce parsing and deserialization costs [4], differential techniques to exploit similarities between messages [17], schema-specific techniques [8,19], table-driven methods [21], and hardware acceleration [7].

On the CPU front, manufacturers are shifting towards using increasing transistor densities to provide multiple cores on a single chip, rather than faster clock speeds. This has led us to investigate the use of parallelization to improve the performance of XML parsing. In previous work, we have explored parallelizing DOM-style parsing [20], where an in-memory DOM tree is generated from an entire document. In many situations, however, SAX-style parsing is preferred [1]. SAX-style parsing is often faster because no in-memory representation of the entire document is constructed. Furthermore, in some situations, the XML itself is streaming, for which an event-based API like SAX is much more suitable. Thus, in this paper, we investigate how SAX parsing can be parallelized. Parallel SAX parsing may seem to be an oxymoron, since SAX is inherently sequential. However, even though the sequence of SAX callbacks (representing the events) are sequential, we show that it is possible to parallelize the parsing computations prior to issuing the callbacks, and only sequentialize just before issuing the callbacks.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 142–156, 2008.

In some applications, application-level processing in the callbacks themselves may dominate the total time. Parallel SAX parsing could alleviate that, however, by leveraging schema information to determine which elements are not ordered (such as those in an `<xsd:all>` group). With applications that are multicore-enabled, callbacks for such elements could be issued concurrently. We do not leverage such information in current work, but see this current work as a necessary precursor to such techniques. Furthermore, we note that XML parsing is also a necessary first step for more complex processing such as XPath or XSLT. Though on a single core, the XML parsing may not dominate this more complex processing, Amdahl's law [2] shows that even relatively short sequential phases can quickly limit speedup as the number of cores increases. Parallel parsing is thus more important than a single core analysis might suggest.

Previous researchers have investigated parallel, depth-first traversal, but with the assumption that the the entire tree was always available [14,15]. Streaming situations, on the other hand, are different because the incoming stream can present itself as a possible source of new work. Furthermore, in a streaming situation, it is important to process work in a depth-first *and left-right* order when possible, so that nodes become no longer needed and can be freed.

A number of previous researchers have also explored parallel XML processing. Our previous work in [9,10,11,12] has explored DOM-style parsing. The work in [13] takes the advantage of the parallelism existing between different XML documents and then structures the XML data accordingly. This approach, however, is focused on processing a certain class of queries, as opposed to XML parsing. The work in [18] uses an intermediary-node concept to realize a workload-aware data placement strategy which effectively declusters XML data and thus obtains high intra-query parallelism. It also focuses on queries, however.

2 Background

Parallelism can be broadly categorized into pipelining, task-parallelism, and data-parallelism, depending on whether the execution units are distributed across different sequential stages in the processing of a single data unit, different independent tasks that may be executed in any order, or different, independent pieces of the input data, respectively.

A pipelined approach to XML parsing would divide the parsing into stages, and assign one core to each stage. Such a software pipeline can have satisfactory performance, but if followed strictly and used in isolation, suffers from inflexibility due to the difficulty of balancing the pipeline stages. Any imbalance in the stages results in some cores remaining idle while other cores catch up. Furthermore, adding a core generally requires redesign and re-balancing, which can be a challenge.[1]

[1] To improve cache locality, cores may be assigned to data units rather to stages. So, a single core might execute all stages for a given data unit. This does not resolve the stage balancing issue, however, except when there are no dependencies between data units, in which case we can simply use data parallelism. If there *are* dependencies between data units, then a core C2 processing data unit D2 in stage 1 must still wait for core C1 to complete data unit D1 in stage 2, before C2 can begin stage 2 processing on D2.

Task parallel approaches divide the processing into separate tasks which can then be performed concurrently. The difference between pipelining and task parallelism is that in the former, the output of one stage is fed into the input of the other, while in the latter, the tasks are independent (though the final output will depend on the completion of all tasks). Task parallelism also suffers from inadaptability to a varying number of cores. The code must be specifically written to use a given number of cores, and taking advantage of another core would require rewriting the code such that an additional task could be assigned to the additional core, which may be difficult and time-consuming. Tasks must be carefully balanced so that no cores are left idle while waiting for other tasks to complete.

In a data-parallel approach, the input data is divided into pieces, and each piece is assigned to a different core. All processing for that piece is then performed by the same core. Applied to streaming XML, each core would parse separate pieces of the XML stream independently. As each core finishes, the results would be spliced back together, in parallel. Adding an additional core would only require reading in an additional chunk for the new core to process. The code does not need to be re-implemented, and thus it is possible to dynamically adjust to a variable number of cores on-the-fly.

Under dynamic conditions as typically encountered in software systems (as opposed to hardware systems that do not change unless physically modified), data parallelism is often more flexible when the number of cores may change dynamically, and may have reduced bus bandwidth requirements. The major issue with data parallelism, however, is that the N pieces must be independent of each other. That is, it must be possible to parse a piece of the data without any information from any other piece. Applied to an incoming XML stream, this poses a problem, since XML is inherently sequential, in the sense that to parse a character at position p, all characters at position 1 to $p - 1$ may first need to be parsed.

2.1 Hybrid Parallelism

We note that in practice there is no need to adhere strictly to a single style of parallelism in a program. Combining the forms, such as pipelined and data parallelism, to form a hybrid can improve flexibility and load balancing. For example, data dependencies can be handled in the first stage of a pipeline by strictly sequential processing. This processing can annotate the output stream of the first stage to address such ordering dependencies. The second stage could then process chunks of the incoming data stream independently in a data-parallel fashion, using the annotations to resolve any ordering dependencies. For such a technique to be beneficial, it must be possible to address data dependencies using a fast sequential stage, and defer more computationally intensive processing to a data-parallel later stage.

The improved flexibility can be illustrated when we consider adding a core to a simplified example. Consider a balanced, two-stage pipeline, where dependencies prevent a datum from entering a stage until the previous datum has completed that stage. Thus, to add a core, a new stage would need to be created, requiring recoding the software while maintaining stage balance. Now assume that the second stage is data parallel, and that the second stage takes much longer for a given datum than the first stage. That is, a datum cannot enter the first stage until the preceding datum has finished the first stage, but a datum can begin the second stage as soon as it has finished the first stage, without waiting for previous datum to finish the second stage. In this case, an additional core can simply be added to the data-parallel second stage. Since this stage is slow relative to the first stage, sufficient data units can be in second stage processing to keep the additional core busy.

Stage balancing is thus no longer required, as long as the data parallel stage is slower than all sequential stages. In a pure pipelined parallelism, stage balancing is required because any stage that executes slower will block (or starve) other stages. The core that is assigned to an idle stage cannot be re-assigned to other stages, because all stages are sequential and so cannot utilize another core. If one of the stages is data parallel, however, any idle core can be assigned to the data parallel stage, as long as the data parallel stage has work to do. This will be true as long as it is the slowest stage.

2.2 Amdahl's Law under Hybrid Parallelism

Amdahl's Law states that if the non-parallelizable part of a particular computation takes s seconds, and the parallelizable part takes p seconds for a single core, then the speedup obtainable for N cores is $(s + p)/(s + p/N)$. Amdahl's Law is a particular perspective from which to view parallel performance. It assumes that there is a single, fixed-size problem, and models the computation as consisting of mutually exclusive sequential phases and parallel phases. It then considers the total time to compute the answer when using a varying number of processors, holding the problem size fixed. It shows that the speedup is dramatically limited, even by relatively small amounts of sequential computation. Fundamentally, the problem is because during the sequential phase, all other processors are idle. The more processors are used, the greater the amount of wasted processing power during this idling.

When considering streaming input, however, the input presents as a continuous stream, rather than a fixed-size, discrete problem. Thus, speedup of the throughput can be a more useful metric than speedup of a fixed-size work unit. If there is no pipelining, then Amdahl's law can be applied to throughput directly. The throughput is inversely proportional to the total time to process any unit of data, and that will simply be the sequential time plus the total parallel computation divided by the number of processors. Without pipelining, when the sequential computation is occurring, all other cores are idle, as in the non-streaming case.

If we consider hybrid parallelism under streaming, however, the picture can appear significantly different. Consider a pipeline consisting of an initial sequential stage followed by a data-parallel stage. While one core is performing the sequential computation, all other cores are working in parallel on the second stage. When cores are added, they are assigned to the second stage, where they always have work, as long as the first stage is not the bottleneck. Assuming that the sequential stage can be made fast, often the core assigned to the first stage will have no work to do. Since the second stage is data-parallel, however, this core can simply be time-sliced between the first stage and the second stage to achieve load balancing.

To illustrate this, in Figure 1 we compare the throughput speedup of a two-stage computation under two different scenarios: pipelined, streaming; and non-streaming, non-pipelined. In this manufactured example, we assume that for a given datum, the

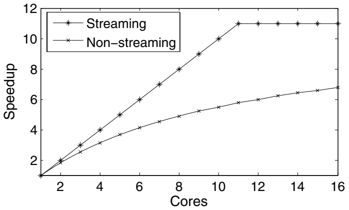


Fig. 1. Throughput speedup of a two-stage computation under streaming, pipelined; and non-streaming, non-pipelined scenarios

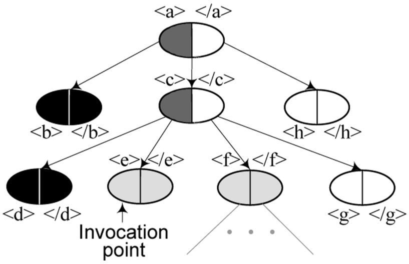


Fig. 2. This shows a conceptual representation of a streaming XML document, in left-to-right order. Elements are shown here as split ovals, representing two nodes. The left half is the start-tag node, and the right half is the end-tag node. Note that the child links actually only point to the start-tag node; end-tag nodes are not actually linked into the tree, but are rather pointed-to by the start-tag node. Nodes in different categories (explained in the text) are shown in different colors.

sequential first stage takes 1 second, and the data-parallelizable second stage takes 10 seconds, for a single core. If a single, discrete problem is presented, and the first stage and the second stage are not pipelined, then the total time per datum is simply $1 + 10/N$, where N is the number of cores. Speedup is then given by $11/(1 + 10/N)$ per a straightforward application of Amdahl's Law. If, however, a stream of data is presented, and we allow the first and second stages to be pipelined, we can assume that one core is time-sliced between the sequential first stage and the data-parallel second stage, and that all other cores are always working on the second stage. If there are 11 cores or fewer, the time taken per datum is then simply $11/N$, since the total computation time for a datum is 11 seconds (for a single core), and no cores are ever idle. The throughput is then given by $N/11$. If there are more than 11 cores, the first stage becomes the bottleneck, and throughput then becomes rate-limited at 1 datum/second by the first stage, which we have assumed cannot be parallelized.

As the graph shows, speedup under hybrid parallelism is limited by the throughput of the slowest sequential stage. Prior to that point, however, the speedup is in fact linear. This observation means that speedup for stream-oriented processing under multicore processing can have significantly different characteristics than the usual non-streaming case, though asymptotically they reach the same limit.

3 Parallel Parsing of Streaming XML

To perform namespace prefix lookup, some kind of data structure is necessary. We could use a stack for this, but a tree is a natural representation that makes the inherent parallelism more explicit, and we chose this model. Thus, we represent start-tags, end-tags, and content as nodes. Only start-tags are actually linked into the tree via parent-child links, since only start-tags need actual namespace prefix lookup. An end-tag is linked-in via a pointer to it in the corresponding start-tag, as well as by the SAX event list discussed below. Content nodes are linked-in via the SAX event list only.

Although the tree structure is required for the namespace prefix lookups, the nodes are also used to represent and maintain SAX events. To facilitate this usage, we also maintain a linked list of the nodes in SAX order (depth-first). Thus, two types of organization are superimposed on each other within a single data structure. The first organization is that of a tree, and is used for namespace prefix lookup. The second is that of a linked list, and is used for the callback invocations used to send SAX events to the application.

The primary challenge with parallel SAX parsing is that when an XML document is streamed in, only part of the tree is resident in memory at any one moment. This resident part consists of open start-tags, closed start-tags that are still needed for namespace prefix lookup, and nodes that have not yet been sent to the application as SAX callbacks. This rooted, partial tree is called the *main* tree. More precisely, we can divide the nodes of a streaming tree into four categories, as shown in Figure 2. The first category is nodes that have not been received at all yet. Nodes `<g>`, `</g>`, `</c>`, `<h>`, `</h>`, and `</a>` are in this category. The second category is nodes that have been received, but have not yet been invoked. Nodes `<e>`, `</e>`, `<f>`, and `</f>` are in this category.

Since prefix-to-URI bindings are stored with the nodes, a node needs to stay resident even after it has been sent to the application, as long as it has descendants that have not yet completed namespace prefix lookup. An open start-tag may still have children on the wire, and so must also stay resident. Note that the root start-tag is only closed at the end of the document, and so is always resident. These types of nodes form the third category, and nodes `<a>` and `<c>` are examples of this. The last category is nodes that are no longer needed in any way. These nodes represent events that have already been sent to the application via callbacks, and will not be needed for further namespace prefix lookups. These nodes can be deleted from memory. Nodes `<b>`, `</b>`, `<d>`, and `</d>` are in this category. In summary, category 2 and 3 nodes are resident in memory, while the rest are either still out on the network, or have been deallocated.

3.1 Stages

Our SAX parsing thus uses a five-stage software pipeline with a combination of sequential and data-parallel stages. Stage one (PREPARSE) reads in the input XML stream by chunks, and identifies the state at the beginning of each chunk via a fast, sequential scan (called a preparse). Stage two (BUILD) then takes the data stream from the PREPARSE stage, and builds fragments of the XML tree in a data-parallel, out-of-order fashion. Each chunk from the PREPARSE stage generates a forest of XML fragments. Stage three (MERGE) sequentially merges these forests into the current main tree to allow namespace prefix lookup to occur. Stage four (LOOKUP) then performs lookup processing on nodes in a parallel, depth-first order (though the actual lookup itself goes up the tree). Finally, in stage five (INVOKE), SAX events are sequentially invoked as callbacks to the application.

This pipelining of sequential and parallel stages into a hybrid parallelism provides greater scheduling flexibility as explained in Section 2.1. The fundamental benefit of hybrid parallelism is that it allows cores that might otherwise be idle due to stage imbalance to be assigned to data-parallel stages, in most cases. Though the PREPARSE, MERGE, and INVOKE stages are still intra-stage sequential, the stages are designed so that as much as computation as possible is moved out of these stages, and into data parallel stages, thus enhancing scalability.

The data unit between the stages changes as the data flows. In stage one, the XML stream is partitioned into chunks, and passed to stage two. In stage two, the chunks are parsed into tree fragments consisting of graph-based nodes. Stage three merges these fragments into the current tree. Stage four can now operate on a tree, and remain mostly oblivious to the fact that it is actually a stream. Stage five operates on a sequence of SAX events.

There is an approximate, but inexact correspondence between the stages and the categories shown in Figure 2 and explained in the previous section. Any node that has been deallocated or not yet received is not in any stage, of course. A node that is in the PREPARSE, BUILD, or LOOKUP stage is in category two, since they have been received, but not yet invoked. A node that has passed the INVOKE stage could be in category three, if it needs to stay resident for children nodes to perform LOOKUP stage; or, it could be deallocated, and thus in category four.

We now describe the stages in more detail.

3.2 Stage One (PREPARSE): Preparse Chunks

To obtain data-parallelism, in the ideal case we could simply divide the incoming XML document into chunks, and then parse each chunk in parallel to build the tree. The problem with this, however, is that we do not know which state to begin parsing each chunk, since the first character is of unknown syntactic role at this point. Thus, a parser that begins parsing at some arbitrary point in an XML stream (without having seen all previous characters) will not know which state in which to begin.

To address this, and still allow parallelism, we use a fast PREPARSE stage to first identify the basic structure of the XML stream. We divide the incoming XML stream into chunks, and then push each chunk through a table-driven DFA that identifies the major syntactic structures within the document. This gives us the state at the end of the chunk, and thus the state at which to begin the next chunk. This initial state is then passed to the next stage, along with the actual chunk data. For example, this DFA will determine if a chunk begins within a start-tag, end-tag, character content, etc. The effect is to “fast-forward” through the XML stream.

Because the preparse process is sequential, this technique works only as long as the PREPARSE stage can be much faster than the actual full parsing. In practice, we have found this to be true. If the preparse itself starts to become a bottleneck, we have shown in previous work how the preparse itself can be parallelized [11].

3.3 Stage Two (BUILD): Build Tree Fragments

The BUILD stage parses chunks in parallel, and builds tree fragments from each chunk, corresponding to the XML fragment in the chunk. The work of the BUILD stage can occur in a data parallel fashion, and so can have multiple cores assigned to it, one for each chunk. It obtains the proper state at which to begin parsing each chunk from the previous PREPARSE stage. The DFA for this stage, called the bt-DFA (Build Tree Fragment DFA), parses the syntactic units in detail. For example, in the start-tag, the element name will be parsed into namespace prefix and local part. The attribute will be parsed into namespace prefix, attribute name's local part, and attribute value.

The bt-DFA is different from the DFA for the preparse stage, because it more precisely identifies the syntactic units in XML. We thus need to map the preparsing DFA state given by the PREPARSE stage to the state of the more detailed bt-DFA.

In the preparsing DFA, a single state 3 is used to identify any part inside a start-tag that is not inside an attribute value. So, inside an element name is treated the same as inside an attribute name. In the bt-DFA, the start-tag is treated in greater detail, so preparser DFA state 3 may map to a number of different states in the bt-DFA. The preparser DFA, however, simply does not provide enough information to know which bt-DFA state should be used for state 3. To solve this ambiguity, we create a special auxiliary state in the bt-DFA, state 0. Preparse state 3 will map to bt-DFA state 0, but as soon as it receives enough input to distinguish further which syntactic unit it is in, it will go to the corresponding bt-DFA state.

In XML documents, start-tags and end-tags are organized in a paired fashion in depth first order. To assist in building the tree fragments, we maintain a stack of start-tag nodes. When we encounter a start-tag, we add it as a child of the top start-tag node, and then push it on to the stack. When we encounter an end-tag, we close the top start-tag, and pop the stack. Although the tree structure is required for the namespace prefix lookups, the SAX event list must also be maintained, as explained near the beginning of Section 3. Therefore, as the DFA progresses, it also creates the event list. Every time a start-tag, end-tag, or character content SAX event is recognized, its corresponding node is linked to the last such node. This creates the list in SAX event order, which is actually the same as XML document order.

For well-formed XML fragments, the start-tags and end-tags inside will be all paired, thus, the stack at the end of parsing such fragment will be empty. But because the XML fragments are partitioned in an arbitrary manner, there will usually be more start-tags than end-tags, or more end-tags than start-tags. If there are more start-tags than end-tags, there will be start-tags remaining on the stack when this stage finishes a chunk. This stack is preserved for merging in the next stage. The output of this stage is thus a set of tree fragments, and a stack remnant, as shown in Figure 3. The nodes in the tree fragments are linked together in a SAX event list in addition to the tree links.

3.4 Stage Three (MERGE): Merge Tree Fragments

The result of the BUILD stage are disconnected tree fragments. These fragments are not connected to the root, or nodes in previous chunks, and so namespace prefix lookups cannot be done. The purpose of the MERGE stage is thus to merge these fragments into the main, existing tree. The merge is performed using the stack from the tree fragment created from the preceding chunk, and the event list of the successor chunk. The merge occurs by traversing the successor fragment event list and matching certain nodes to the predecessor stack, but in such a manner that only the top edge of the successor fragment is traversed, since only the top edge of the successor fragment needs to be merged. The internal portion of the successor fragment is already complete. The algorithm for

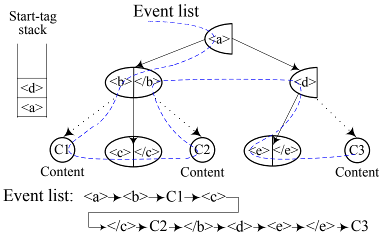


Fig. 3. This diagram shows a tree fragment created by the BUILD stage. Solid-line arrows show actual child links. Dotted line arrows show conceptual child relationships that are not embodied via actual pointers. The dashed line shows how the nodes are linked into an event list for this fragment. The start-tag stack holds unclosed start-tags, to be closed in later fragments.

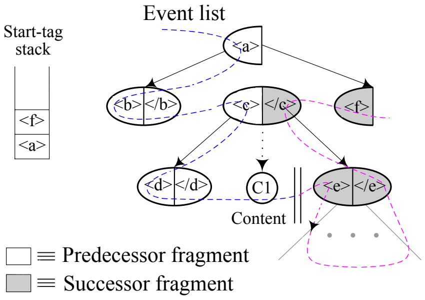


Fig. 4. This diagram shows two tree fragments immediately after they have been merged, including the stack after the merge. Nodes `<e>`, `</e>`, `</c>`, and `<f>` were in the successor fragment, while the other nodes were in the preceding fragment. The double vertical line between nodes `c1` and `<e>` shows the division between the event list of the predecessor tree fragment from the event list of the successor tree fragment. If successor fragment is not the last fragment in the document, some start-tags will be unclosed, such as `<a>` and `<f>` in this example.

merging is given in Algorithm 1. Figure 4 shows two tree fragments after they have been merged.

In addition, the chunking process of the PREPARSE stage creates chunk boundaries which fall at arbitrary character positions. Thus, some XML syntactic units, such as an element name, will be split across two chunks. The MERGE stage also resolves these split units, completing them so that their corresponding nodes can be processed normally in later stages.

3.5 Stage Four (LOOKUP): Namespace Prefix Lookup

The purpose of this stage is to lookup namespace prefixes. The actual lookup is performed by going up the tree to the root. But the order in which nodes undergo lookup processing is top-down. In other words, a child cannot undergo lookup processing unless its parent has already undergone the LOOKUP stage. Thus, lookup processing is ordered using a parallel depth-first traversal of the main tree.

To assist in load-balancing, work queues of nodes that should next be traversed are maintained. These queues essentially contain nodes that are at the "traversal front". Each thread maintain a work queue of rooted sub-trees, represented by the root node of the sub-tree. As each sub-tree root is processed, each child of this root then becomes the root of another, new sub-tree, and so is added to the thread-local work queue. Because new tree fragments are being merged into the main tree concurrently with this traversal, it is possible that after a node is processed, there are still unmerged, "future" children. These children would not be added to the local work queue, because they were not

```
Algorithm 1: Merge two tree fragments
Data: Tree fragment $prev$, Tree fragment $next$
Result: The merged tree fragment into $prev$
EventNode $p \leftarrow next.event\_list.head$;
while $p$ exists AND $prev.stack \neq \emptyset$ do
    if is_StartTag(p) then
        Set $p.parent$ refer to $prev.stack.top()$;
        Link $p$ as a child node of $p.parent$;
        if $p.EndTag$ exists then
            $p \leftarrow p.EndTag$;
            Set $p.parent$ refer to $prev.stack.top()$;
        else
            break;
    else if is_EndTag(p) then
        StartElement $s \leftarrow prev.stack.top()$;
        Set $s.EndTag$ refer to $p$;
        Set $p.parent$ refer to $s.parent$;
        Pop $prev.stack$;
    $p \leftarrow p.next\_event$;
Stack $next.stack$ onto $prev.stack$;
Concatenate $prev.event\_list$ with $next.event\_list$;
```

Fig. 5. The algorithm for merging two tree fragments

linked in as children when the parent was processed, and thus would never be processed by the LOOKUP stage.

To address this, when a node is merged into the main tree, a check is performed on the parent node. If the node has already undergone namespace prefix lookup, then the node is instead added to a global, unassigned work queue. When a thread runs out of work, it first checks the unassigned work queue, and if non-empty, it grabs it as its new work.

3.6 Stage Five (INVOKE): Invoke Callbacks

Finally, the actual SAX events must be sent to the application by invoking callbacks. We assume in this paper that the callbacks must be issued sequentially, though it is possible that a multi-core enabled application could handle concurrent callbacks in schema types that are order independent, such as `<all>`. One way of invoking callbacks would be to have a single thread invoke them. This would result in poor cache locality, however, since this thread would likely not belong to the core that last accessed the node currently being invoked.

To improve locality, we utilize a virtual token that is passed along the SAX event chain, called the NEXT token. This token is implemented via a flag. To pass the token from an earlier node to a later node, the flag is set in the later node and cleared in the earlier node. Chained locking is used to prevent race conditions.

When a thread finishes namespace lookup on a node, it checks to see if it is the next node, by virtue of holding the token. If it is, it invokes it, and passes the token to the next node in the chain, and continues to invoke it also, if it has been completed and is ready to be invoked. The chain continues until encountering a node that is not ready to be invoked. Though multiple threads could be in the LOOKUP stage at the same time, the mechanism used to pass the token down the chain ensures that the INVOKE stage is sequential.

Another way of doing something similar would be have a special global pointer that always points to the next node to be invoked. This would require that all cores continuously read and update the same pointer, however. By using the NEXT token, we avoid excessive operations to the same memory location, avoiding cache-line ping-pong.

3.7 Execution

One way of implementing the stages would be to start a set of threads for every stage, with each sequential stage having just one thread. Parallel stages would have as many threads as there are cores. This architecture relies on the OS to perform all scheduling, however. Furthermore, as work moves along the pipeline, it may jump from one core to another, resulting in poor cache locality.

To address this, another way of implementing the stages is to represent each stage as a set of work units, and is what we use. Each thread then examines each stage, and decides what work to pick up. Once a thread accepts work, it can perform more than one stage on the same data, improving cache locality. In other words, rather than cores exhibiting affinity to stages, cores exhibit affinity to *data*, and so the core and the data move together through the stages.

At the beginning of execution, a thread is started for every core. It then executes a main loop that examines each stage, deciding which stage has work that should be done, and of those, which stage has the highest priority work. It first checks to see whether or not there is any unassigned LOOKUP stage work to perform. If so, it accepts that work. If not, it then checks to see whether or not there is any new data to be read from the network and pushed through the PREPARSE stage. After pushing the data through the PREPARSE stage, the same thread will execute BUILD and MERGE on it. All LOOKUP stage work is initiated through the unassigned work list. INVOKE stage work is initiated internally by the LOOKUP stage itself. If there is no work in the main loop, work stealing is done from a randomly chosen victim. We currently only steal a single sub-tree from a thread at a time, rather than half of the remaining work.

Thus, this is an explicit form of scheduling, which gives us control over what work to do next. Note that if we simply had a set of threads for each stage, the OS may decide to run a thread which is runnable, but may not be best choice for temporal cache locality or other efficiency reasons.

3.8 Memory Management

Memory management is a challenging problem in parallel programs. The complex, dynamic data structures needed for load-balancing and concurrency, complicate determining exactly when an object can be freed. To handle this, we have adopted the commonly used technique of reference counting. In this technique, a count is maintained with every object, indicating the number of pointers that are currently pointing to it. When

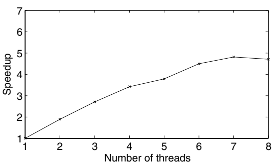


Fig. 6. Speedup of the parallel SAX parser

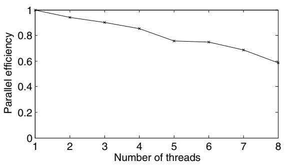


Fig. 7. Efficiency of the parallel SAX parser

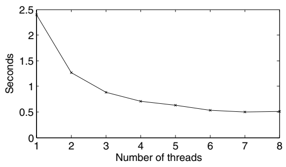


Fig. 8. Total parse time

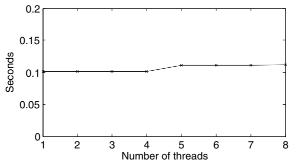


Fig. 9. Time used by the PREPARSE stage over the document

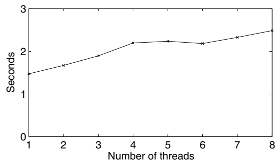


Fig. 10. Time used by the BUILD stage summed over all threads over the document

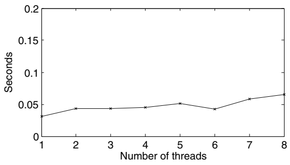


Fig. 11. Time used by the MERGE stage over the document

the count goes to zero, this means that the object is no longer referenced, and can be freed. Atomic increment and decrement operations are used to reduce the cost of locking needed to maintain the counts.

Another issue is thread-optimized memory allocation. This has been the subject of significant research, and since our goal in this paper is not to address this separate topic, we use a simple, free list that we maintain separately for each thread. Thread specific data is used to keep the bookkeeping for each thread separate from the others, thus obviating the need for expensive locking.

4 Performance Results

Our experiments were conducted on a 8-core Linux machine with two Intel Xeon L5320 CPUs at 1.86 GHz. Each test was run five times and the first measurement was discarded, so as to measure performance with the file data already cached. The remaining measurements were averaged. The programs were compiled with `g++` 4.0 with the option `-O`. The SAX callbacks fulfilled by our hybrid parallel SAX parser are based on the SAX2 [1,3] API, through a C++-style binding. The test file is a file named `1kzk.xml` which contains molecular information representing the typical shape of XML documents. We obtained this XML document from the Protein Data Bank [16]. We varied the number of atoms in the document to create a 34 MB XML document.

The speedup of our parallel SAX parser is shown in Figure 6. The speedup is compared to our parallel SAX parser running with one thread. We obtain reasonable speedup up to about six or seven cores. The corresponding wall clock time is shown in Figure 8. To get a better sense of scalability, we plot the parallel efficiency in Figure 7.

To investigate where time was being spent, we also graphed the time spent in the PREPARSE, BUILD, and MERGE stages, in Figures 9, 10, and 11, respectively.

Measurements for these were done using using the real-time cycle counter based clock available on Intel CPUs. The BUILD time is summed over all threads, and so is greater than the total wall clock time given in Figure 8. The PREPARSE stage is currently sequential, and takes a relatively small amount of the total time. We note that the time spent in this stage does not have the same type of impact that would result from using this directly in an Amdahl's Law calculation, as explained in Section 2.1. This is because this PREPARSE stage is pipelined with other stages. Eventually, however, further improvements in scalability will need to address the PREPARSE stage. In previous work [11], we have shown how to parallelize the preparsing, and so this work would need to be incorporated at some point. The MERGE stage is currently sequential, but could actually be parallelized. That is, it is possible to merge fragment F1 and F2 at the same time that F2 is being merged with F3. We note that the MERGE stage takes a relatively small fraction of the total time, however, so parallelizing it might be of somewhat lower priority.

To investigate how to improve scalability further, we conducted additional timing tests, and manually instrumented the code. Part of the problem is due simply to load balancing. Some thread convoy effects are causing some periods of idle threads, which are exacerbated when the number of cores are high. Better work-stealing, better scheduling, and perhaps introducing feedback though control-theoretic techniques can improve this. Another problem is that large XML documents tend to be broad, and thus have an upper-level node that has a large number of children. This results in a hot-spot on the child list of that node, and also the reference count. This can partially be seen by the fact that even though the total amount of work remains constant, the time taken by the BUILD stage increases as the number of threads increases. Future work will address this by using shadow nodes to turn long children lists into trees of descendants, thereby reducing hot-spots. The shadow nodes will help disperse the hot spots, but will simply be skipped during callbacks.

5 Conclusion

To improve memory usage and performance, streaming XML applications are commonplace. For these, SAX-style parsing is the natural choice due to the stream-orientation of event-based callbacks. However, since XML is inherently sequential, parallelization of SAX-style parsing is challenging. To address this, we have devised a parallel, depth-first traversal technique for streaming trees that is effective in parallel, SAX-style parsing of XML. We have shown how this can be used in a five-stage hybrid pipeline which effectively extracts data parallelism from the parsing process, allowing multiple cores to work concurrently within those stages. The sequential requirements of the SAX parsing will be fulfilled by sequential stages. Since the major computational work are done by the data parallel stages, we are able to achieve good performance gain. The design of this approach allows the flexible utilization of a varying number of cores.

Using this approach, we show scalability up to about 6 or 7 cores. As manufacturers increase the number of cores, our future work will seek to further reduce the sequential stages, and thus exploit future CPUs. Techniques such as lock-free synchronization may also be employed to reduce synchronization costs. We note that when considering the scalability of processing that has not been parallelized by previous research, the alternative is not to use parallelism at all, which would result in no speedup. For applications that can benefit from faster SAX performance, cores may end up being wasted.

References

1. SAX, http://www.saxproject.org/
2. Amdahl, G.M.: Validity of the single-processor approach to achieving large scale computing capabilities. In: Proceedings of AFIPS Conference, Atlantic City, NJ, vol. 30, pp. 483–485 (1967)
3. Brownell, D.: SAX2. O'Reilly & Associates, Inc., Sebastopol (2002)
4. Chiu, K., Devadithya, T., Lu, W., Slominski, A.: A Binary XML for Scientific Applications. In: International Conference on e-Science and Grid Computing (2005)
5. Chiu, K., Govindaraju, M., Bramley, R.: Investigating the limits of soap performance for scientific computing. In: HPDC 2002 (2002)
6. Head, M.R., Govindaraju, M., van Engelen, R., Zhang, W.: Benchmarking xml processors for applications in grid web services. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089. Springer, Heidelberg (2006)
7. IBM. Datapower, http://www.datapower.com/
8. Kostoulas, M.G., Matsa, M., Mendelsohn, N., Perkins, E., Heifets, A., Mercaldi: Xml screamer: an integrated approach to high performance xml parsing, validation and deserialization. In: WWW 2006: Proceedings of the 15th international conference on World Wide Web, NY, USA (2006)
9. Lu, W., Pan, Y., Chiu, K.: A Parallel Approach to XML Parsing. In: The 7th IEEE/ACM International Conference on Grid Computing (2006)
10. Pan, Y., Lu, W., Zhang, Y., Chiu, K.: A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs. In: 7th IEEE International Symposium on Cluster Computing and the Grid, Brazil (May 2007)
11. Pan, Y., Zhang, Y., Chiu, K.: Simultaneous Transducers for Data-Parallel XML Parsing. In: 22nd IEEE International Parallel and Distributed Processing Symposium, Miami, Florida, USA, April 14-18 (2008)
12. Pan, Y., Zhang, Y., Chiu, K., Lu, W.: Parallel XML Parsing Using Meta-DFAs. In: 3rd IEEE International Conference on e-Science and Grid Computing, India (December 2007)
13. Qadah, G.: Parallel processing of XML databases. In: Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering, May 2005, pp. 1946–1950 (2005)
14. Rao, V.N., Kumar, V.: Parallel depth first search. part i. implementation. Int. J. Parallel Program. 16(6), 479–499 (1987)
15. Reinefeld, A., Schnecke, V.: Work-load balancing in highly parallel depth-first search. In: Proc. 1994 Scalable High-Performance Computing Conf., pp. 773–780. IEEE Computer Society, Los Alamitos (1994)
16. Sussman, J.L., Abola, E.E., Manning, N.O.: The protein data bank: Current status and future challenges (1996)
17. Takase, T., Miyashita, H., Suzumura, T., Tatsubori, M.: An adaptive, fast, and safe xml parser based on byte sequences memorization. In: WWW 2005: Proceedings of the 14th international conference on World Wide Web, pp. 692–701. ACM Press, New York (2005)
18. Tang, N., Wang, G., Yu, J.X., Wong, K.-F., Yu, G.: Win: an efficient data placement strategy for parallel xml databases. In: 11th International Conference on Parallel and Distributed Systems (ICPADS 2005), pp. 349–355 (2005)

19. van Engelen, R.: Constructing finite state automata for high performance xml web services. In: Proceedings of the International Symposium on Web Services (ISWS) (2004)
20. W3C. Document Object Model (DOM), http://www.w3.org/DOM/
21. Zhang, W., van Engelen, R.: A table-driven streaming xml parsing methodology for high-performance web services. In: IEEE International Conference on Web Services (ICWS 2006), pp. 197–204 (2006)

Performance Analysis of Multiple Site Resource Provisioning: Effects of the Precision of Availability Information

Marcos Dias de Assunção[1,2] and Rajkumar Buyya[1]

[1] Grid Computing and Distributed Systems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
[2] NICTA Victoria Research Laboratory
The University of Melbourne, Australia
{marcosd,raj}@csse.unimelb.edu.au

Abstract. Emerging deadline-driven Grid applications require a number of computing resources to be available over a time frame, starting at a specific time in the future. To enable these applications, it is important to predict the resource availability and utilise this information during provisioning because it affects their performance. It is impractical to request the availability information upon the scheduling of every job due to communication overhead. However, existing work has not considered how the precision of availability information influences the provisioning. As a result, limitations exist in developing advanced resource provisioning and scheduling mechanisms. This work investigates how the precision of availability information affects resource provisioning in multiple site environments. Performance evaluation is conducted considering both multiple scheduling policies in resource providers and multiple provisioning policies in brokers, while varying the precision of availability information. Experimental results show that it is possible to avoid requesting availability information for every Grid job scheduled thus reducing the communication overhead. They also demonstrate that multiple resource partition policies improve the slowdown of Grid jobs.

1 Introduction

Advances in distributed computing have resulted in the creation of computational Grids. These Grids, composed of multiple resource providers, enable collaborative work and resource sharing amongst groups of individuals and organisations. These collaborations, widely known as Virtual Organisations (VOs) [1], require resources from multiple computing provider sites, which are generally clusters of computers managed by queueing-based Resource Management Systems (RMSs), such as PBS and Condor.

Emerging deadline-driven Grid applications require access to several resources and predictable Quality of Service (QoS). A given application may require a number of computing resources to be available over a time frame, starting at a

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 157–168, 2008.

specific time in the future. However, it is difficult to provision resources to these applications due to the complexity of providing guarantees about the start or completion times of applications currently in execution or waiting in the queue. Current RMSs generally use optimisations to the first come first served policy such as backfilling [2] to reduce the scheduling queue fragmentation, improve job response time and maximise resource utilisation. These optimisations make it difficult to predict the resource availability over a time frame as the jobs' start and completion times are dependent on resource workloads.

To complicate the scenario further, users may access resources via mediators such as brokers or gateways. The design of gateways that provision resources to deadline-driven applications relying on information given by current RMSs may be complex and prone to scheduling decisions that are far from optimal. Moreover, it is not clear how gateways can obtain information from current RMSs to provision resources to QoS demanding applications. Existing work on resource provisioning in Grid environments has used conservative backfilling wherein the fragments of the scheduling queue are given to be provisioned by a broker [3]. These fragments are also termed availability information or free time slots. We consider impractical to request the free time slots from providers upon the scheduling of every job due to potential communication overhead.

In this paper, we investigate how the precision of availability information affects resource provisioning in multiple site environments. In addition, we enhance traditional schedulers, allowing the obtention of availability information required for resource provisioning. We evaluate the reliability of the provided information under varying conditions by measuring the number of provisioning violations. A violation occurs when the information given by the resource provider turns out to be incorrect when it is used by the gateway. Additionally, we evaluate the impact of provisioning resources to Grid applications on providers' local requests by analysing the job bounded slowdown. We investigate whether EASY backfilling [4] and multiple partition policies provide benefits over conservative backfilling if job backfilling is delayed, enabling large time slots to be provided to the gateway.

2 Related Work

The performance analysis and the policies proposed in this work are related to previous systems and techniques in several manners.

Modelling providers' resource availability: AuYoung *et al.* [5] consider a scenario wherein service providers establish contracts with resource providers. The availability information is modelled as ON/OFF intervals, which correspond to off-peak and peak periods respectively. However, they do not demonstrate in practice how this information can be obtained from RMSs.

Advance reservations and creation of alternatives to rejected requests: Mechanisms for elastic advance reservations and generation of alternative time slots for advance reservation requests have been proposed [6,7]. These models can be incorporated in the provisioning scenario described in this work to improve resource utilisation and generate alternative offers for provisioning violations. However, we aim to reduce the interaction between resource providers and gateways by allowing the providers to inform the gateways about their spare capacity. We focus on how the availability information can be obtained from RMSs and how reliable it is under different conditions.

Multiple resource partition policies: Work on multiple resource partitions and priority scheduling has shown to reduce the job slowdown compared to EASY backfilling policies [8]. We build on this effort and extend it to enable other multiple partition policies. We also propose a new multiple resource partition policy based on load forecasts for resource provisioning.

Resource allocation in consolidated centres: Padala *et al.* [9] apply control theory to address the provision of resources to multi-tier applications in a consolidated data centre. Garbacki and Naik [10] consider a scenario wherein customised services are deployed on virtual machines which in turn are placed into physical hosts. Although the provisioning of resources to applications in utility data centres is important, here we focus on traditional queue-based RMSs.

Resource provisioning: Singh *et al.* [3,11] present a provisioning model where Grid sites provide information on the time slots over which sets of resources are available. The sites offer their resources to the Grid in return for payments, thus they present a cost structure consisting of fixed and variable costs over the resources provided. The main goal is to find a subset of the aggregated resource availability, termed as resource plan, such that both allocation costs and application makespan are minimised. Our work is different in the sense that we investigate multiple approaches to obtain availability information and how reliable this information can be in multiple site environments.

3 Multiple-Site Resource Provisioning

The multiple site scenario is depicted in Figure 1, which shows DAS-2's configuration used later in the experiments. A Resource Provider (RP) contributes a share of computational resources to a Grid in return for regular payments. An RP has local users whose resource demands need to be satisfied, yet it delegates provisioning rights over spare resources to an InterGrid Gateway (IGG) by providing information about the resources available in the form of free time slots. A free time slot describes the number of resources available, their configuration and time frame over which they will be available. The delegation can be made through a secure protocol such as SHARP [12].

A Grid can have peering arrangements with other Grids managed by IGGs and through which they co-ordinate the use of resources. This work does not address peering arrangements [13]. Here, we investigate how an IGG can provision resources to applications based on the availability information given by RPs.

Problem Description: An IGG attempts to provision resources to meet its users' QoS demands, improve the job slowdown and minimise the number of

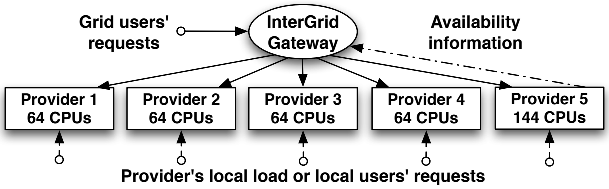


Fig. 1. Resource providers contribute to the Grid but have local users

violations. A violation occurs when a user tries to use the resources allocated by the IGG and they are no longer available due to wrong or imprecise availability information given by the RP. RPs, on the other hand, are willing to increase the resource utilisation without compromising their local users requests. IGG should achieve allocations that minimise the response time and slowdown of Grid users' requests without perceivable impact on the slowdown of the RPs' local requests.

Grid Requests: A request received by an IGG is contiguous and needs to be served with resources from a single resource provider. They contain a description of the required resources and the time duration over which they are required. A request can demand either QoS or a best effort service. A QoS constrained request has an execution estimate, a deadline and a ready time before which the request is not available for scheduling. A best effort job has an execution time estimate but does not have a deadline.

4 Policies Investigated

We have extended traditional scheduling policies in order to obtain the free time slots from resource providers. The policies utilise an 'availability profile' similar to that described by Mu'alem and Feitelson [2]. The availability profile is a list whose entries describe the CPUs available at particular times in the future. These entries correspond to the completion or start times of jobs and advance reservations. By scanning the availability profile and using other techniques described here, the resource providers inform the gateway about the free time slots; the gateway in turn can carry out provisioning decisions based on this information.

Conservative Backfilling Based Policies: Under conservative backfilling, a job is used to backfill and start execution earlier than expected, given that it does not delay any other job in the scheduling queue [2]. In order to reduce complexity, the schedule for the job is generally determined at its arrival and the availability profile is updated accordingly. Given those conditions, it is possible to obtain the free time slots by scanning the availability profile. This approach, depicted in Figure 2a, was also used by Singh *et. al* [3,11]. In that case, the availability profile is scanned until a given time horizon thus creating windows of availability or free time slots; the finish time of a free time slot is either the finish time of a job in the waiting queue or the planning horizon. We have also implemented

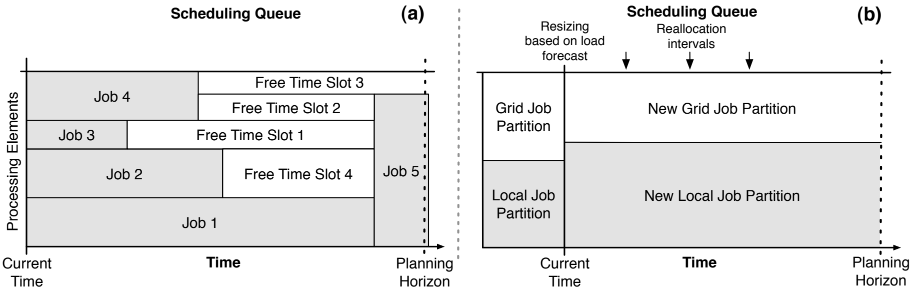


Fig. 2. Obtaining free time slots: (a) conservative backfilling, (b) multiple partitions

a conservative backfilling policy that uses multiple resource partitions based on the EASY backfilling proposed by Lawson and Smirni [8].

Multiple Resource Partition Policies: We have implemented 3 policies based on multiple resource partitions. In our implementation, each policy divides the resources available in multiple partitions and assigns jobs to these partitions according to partition predicates. A partition can borrow resources from another when they are not in use by the latter and borrowing is allowed by the scheduler. One policy implements the EASY backfilling (also termed as aggressive backfilling) described by Lawson and Smirni [8]. In this case, each partition uses aggressive backfilling and has a pivot, which is the first job in the waiting queue for that partition. A job belonging to a given partition can start its execution if it does not delay the partition's pivot and the partition has enough resources. In case the partition does not have enough resources, the job can still start execution if additional resources can be borrowed from other partitions without delaying their pivots. Additionally, the policy uses priority scheduling wherein the waiting queue is ordered by priority when the scheduler is backfilling. In order to evaluate this policy, we attempt to maintain the configuration provided by Lawson and Smirni [8], which selects partitions according to the jobs' runtimes. The partition $p \in \{1, 2, 3\}$ for a job is selected according to Equation 1, where t_r is the job's runtime in seconds.

$$p = \begin{cases} 1, & 0 < t_r < 1000 \\ 2, & 1000 \leqslant t_r < 10000 \\ 3, & 10000 \leqslant t_r \end{cases} \tag{1}$$

We also introduce a new policy, depicted in Figure 2b, in which, the partitions are resized by the scheduler at time intervals based on a load forecast computed from information collected at previous intervals. As load forecasts are prone to be imprecise, when the scheduler resizes partitions, it also schedules reallocation events. At a reallocation event, the scheduler evaluates whether the load forecast has turned out to be an underestimation or not. If the load was underestimated, the policy resizes the partitions according to the load from the last resizing period until the current time and backfill the jobs, starting with the local jobs.

Algorithm 1. Provider's load forecasting policy

```
1  procedure getFreeTimeSlots()
2  begin
3      set number of pivots of local and Grid partitions to $\infty$
4      schedule / backfill jobs in the waiting queue
5      set number of pivots of local and Grid partitions to 1
6      $actualLoad \leftarrow$ load of waiting/running jobs
7      $forecast \leftarrow$ get the load forecast
8      $percToProvide \leftarrow min\{0, 1 - actualLoad\}$
9      $slots \leftarrow$ obtain the free time slots
10     $slots \leftarrow$ resize slots according to $percToProvide$
11     if $percToProvide > 0$ then
12         inform gateway about $slots$ and schedule reallocation event
13     schedule next event to obtain free time slots
14 end
15 procedure reallocationEvent()
16 begin
17     $localLoad \leftarrow$ obtain the local load
18     $forecast \leftarrow$ get the previously computed forecast
19     if $localLoad > forecast$ then
20         set number of pivots of local partition to $\infty$
21         schedule / backfill jobs in the waiting queue
22         set number of pivots of Grid partition to $\infty$
23         schedule / backfill jobs in the waiting queue
24         $slots \leftarrow$ obtain the free time slots
25         inform gateway about $slots$
26     else
27         schedule next reallocation event
28 end
```

We use EASY backfilling with configurable maximum number of pivots, similarly to MAUI scheduler [14]. The policy can be converted to conservative backfilling by setting the number of pivots to a large value, here represented by ∞.

Algorithm 1 describes two procedures of the load forecast policy; *getFreeTimeSlots* is invoked when the provider needs to send the availability information to the IGG whereas *reallocationEvent* is triggered by *getFreeTimeSlots* to verify whether the previous forecast has turned out to be precise or if a reallocation is required. From line 1 to 1 the scheduler becomes conservative backfilling based by setting the number of pivots in each partition to ∞. It also schedules the jobs in the waiting queue. After that, the scheduler returns to EASY backfilling (line 1). Then, from line 1 to 1, the scheduler obtains the load forecast and the free time slots, which are resized by modifying the number of CPUs according to the number of resources expected to be available over the next interval. Next, the scheduler triggers a reallocation event. At line 1 the scheduler verifies whether the forecast was underestimated. If that is the case, it turns the policy to conservative backfilling and informs the gateway about the availability. We have also implemented a multiple resource partition policy based on conservative backfilling.

The policies we consider for the gateway are described as follows:

- **Least loaded resource:** The gateway submits a job to the least loaded resource based on utilisation information sent by the resource providers every ten minutes.

- **Earliest start time:** This policy is employed for best effort and deadline constrained requests when the resource providers are able to inform the gateway about the free time slots. When scheduling a job using this policy, the scheduler is given the provider's availability information and the job. If the providers send the information at regular time intervals, this information is already available at the gateway; otherwise, the gateway requests it from the resource providers. If the job is not deadline constrained, the gateway selects the first provider and submits the job to it. When the job is deadline constrained, the gateway attempts to make a reservation for it. If the reservation cannot be accepted by the provider, the provider updates its availability information at the gateway.

5 Performance Evaluation

5.1 Scenario Description

We have modelled DAS-2 Grid [15] configuration because job traces collected from this Grid and its resources' configuration are publicly available and have been previously studied [16]. As depicted beforehand in Figure 1, DAS-2 is a Grid infrastructure deployed in the Netherlands comprising 5 clusters. The evaluation of the proposed mechanism is performed through simulation by using a modified version of GridSim.[1] We resort to simulation as it provides a controllable environment and enables us to carry out repeatable experiments.

The resource providers' workloads have been generated using Lublin and Feitelson [17]'s model, here referred to as Lublin99. Lublin99 has been configured to generate four month long workloads of type-less jobs (i.e. we do make distinctions between batch and interactive jobs); the maximum number of CPUs used by the generated jobs is set to the number of nodes in the clusters. Grid jobs' arrival rate, number of processors required and execution times are modelled using DAS-2 job trace available at the Grid Workloads Archive.[2] We use the interval from the 9th to the 12th month. The jobs' runtimes are taken as runtime estimates. Although this generally does not reflect the reality, it provides the basis or bounds for comparison of scheduling approaches [18].

To eliminate the simulations' warm up and cool down phases from the results, the last simulated event is the arrival of the last job submitted in any of the workloads and we discard the first two weeks of the experiments. In the case of the forecast based policy, the second week is used as training period. We select randomly the requests that are deadline constrained. In order to generate the request deadlines we use a technique described by Islam *et al.* [19], which provides a feasible schedule for the jobs. To obtain the deadlines, we perform the experiments by using the same Grid environment using aggressive backfilling at the resource providers and 'submit to the least loaded resource' policy at the gateway. A request deadline is the job completion under this scenario multiplied

[1] More information available at: http://www.gridbus.org/intergrid/gridsim.html

[2] Grid Workloads Archive website: http://gwa.ewi.tudelft.nl/pmwiki/

by a *stringency factor*. The load forecasting uses a weighted exponential moving average [20], considering a window of 25 intervals.

Performance Metrics: One of the metrics considered is the bounded job slowdown (*bound*=10 seconds) hereafter referred to as job slowdown for short [18]. Specifically, we measure the bounded slowdown improvement ratio $\mathcal{R}$ given by Equation 2, where s_{base} is the job slowdown using a base policy used for comparison; and s_{new} is the job slowdown given by the policy being evaluated. We calculate the ratio $\mathcal{R}$ for each job and then take the average. The graphs presented in this section show average ratios.

$$\mathcal{R} = \frac{s_{base} - s_{new}}{min(s_{base}, s_{new})} \tag{2}$$

We also measure the number of violations and messages exchanged between providers and IGG to schedule Grid jobs. The reduction in the number of messages required is used for estimating the tradeoff between precision of information and communication overhead. A given job j faces a violation when the inequality $j_{pst} - j_{gst} > T$ is *true*, where j_{gst} is the job start time assigned by the gateway based on the free time slots given by providers; j_{pst} is the actual job start time set by the provider's scheduler; and T is a tolerance time. The experiments performed in this work use a T of 20 seconds. A violation also occurs when a resource provider cannot accept a reservation request made by the gateway.

Policy Acronyms: Due to space limitations, we abbreviate the name of the evaluated policies in the following manner. A policy name comprises two parts separated by +. The first part represents the policy employed by the provider whereas the second is the gateway policy. In the resource provider's side, **Ar** stands for Advance reservation, **Eb** for EASY backfilling, **Cb** for Conservative backfilling, **M** for Multiple partitions and **Mf** for Multiple partitions with load forecast. For the gateway's policy, **least-load** means 'submit to least loaded resource', **earliest** represents 'select the earliest start time', **partial** indicates that providers send free time slot information to the gateway on a periodical basis and **ask** means that the gateway requests the free time slot information before scheduling a job. For example, **ArEbMf+earliest-partial** indicates that providers support advance reservation, EASY backfilling, multiple partitions and load forecasts, whereas the gateway submits jobs selecting the earliest start time based on the availability information sent by providers at regular intervals.

5.2 Experimental Results

The first experiment measures the number of messages required by the policies supporting advance reservation and conservative backfilling (i.e. ArCb), both that request the free time slots and those in which the time slots are informed by providers at time intervals. We investigate whether we can reduce the number of messages required by making the resource providers publish the availability information at gateways at time intervals. We vary the interval for providing the availability information; we also measure the number of violations and average job slowdown to check the tradeoff between the precision of scheduling decisions and the freshness of the information. The planning horizon is set to ∞, so that a provider always informs all the free time slots available. In addition, we consider a two phase commit protocol for advance reservations. The time interval for providing the time slots to the gateway is described in the last part of the name of the policies (e.g. 15 min., 30 min.). The stringency factor is 5 and around 20% of the Grid requests are deadline constrained.

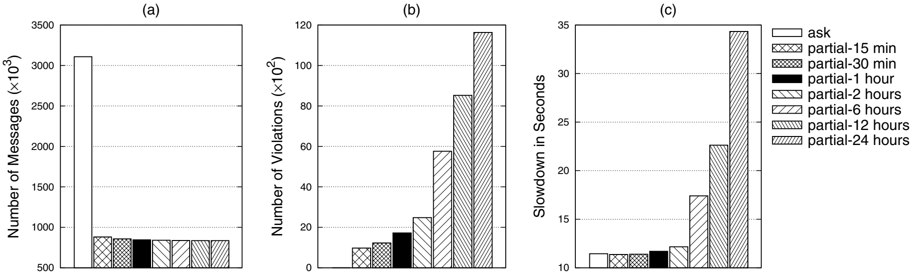


Fig. 3. ArCb+earliest-* policies: (a) number of messages; (b) number of violations; and (c) average job slowdown

Figure 3a shows that the number of messages required by the policy in which the gateway asks for the time slots upon the schedule of every job (i.e. ArCb+earliest-ask) is larger compared to other policies. In contrast, policies that provide the free time slots at regular intervals or when an advance reservation request fails lead to a lower number of messages.

The number of violations increases as the providers send the availability information at larger intervals (Figure 3b). If the scheduling is made based on information provided every 15 minutes, the number of violations is 973, which accounts for 0.43% of the jobs scheduled. To evaluate whether these violations have an impact on the resource provisioning for Grid jobs, we measure the average bounded slowdown of Grid jobs (Figure 3c). As shown in the figure, there is an increase in the job slowdown as the interval for providing the free time slots increases. However, when the providers send availability information every 15 minutes, the average slowdown is improved. We can conclude that for a Grid like DAS-2 wherein providers send the availability information at intervals of 15 to 30 minutes resource provisioning can be possible using a simple policy supporting conservative backfilling.

The second experiment measures the average of jobs ratio $\mathcal{R}$ described in Equation 2 proposed by Lawson and Smirni [8]. The values presented in the graphs are averages of 5 simulation rounds, each with different workloads for providers' local jobs. The set of policies used as basis for comparison are EASY backfilling in the providers and 'submit to the least loaded resource' in the gateway. This way, the experiment measures the average improvement ratio wherein the base policies are EASY backfilling and submit to the least loaded resource.

The resource providers send the availability information to the gateway every two hours. In this experiment we do not consider deadline constrained requests, as they could lead to job rejections by some policies, which would then impact on the average bounded slowdown.

The results conservative backfilling and 'least loaded resource' policies (i.e. ArCb+least-load and ArCbM+least-load) tend to degrade the bounded slowdown of Grid jobs (Figure 4a). The reason is that submitting a job to the least loaded resource, wherein utilisation is computed by checking how many CPUs are in use at the current time, does not ensure immediate start of a job because other jobs in the waiting queue may have been already scheduled. Moreover, the gateway is not aware of the time slot the job will in fact utilise.

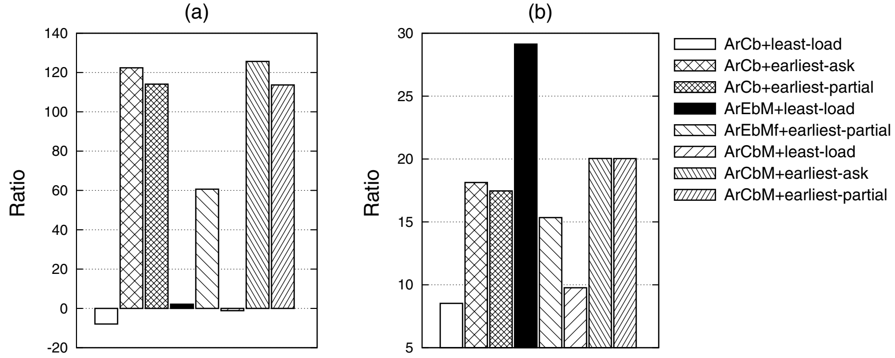


Fig. 4. Slowdown improvement ratio: (a) Grid jobs and (b) providers' local jobs

The multiple resource partition policies with conservative backfilling without priorities and providing the free time slots to the gateway improve the average slowdown of both Grid jobs (Figure 4a) and providers' local jobs (Figure 5b). ArEbM+least-load, proposed by Lawson and Smirni [8], improves the slowdown of local jobs (Figure 4b) providing little changes in the slowdown of Grid jobs. That occurs because in the original implementation of this policy, higher priority is given to local jobs. The EASY backfilling policy that resizes the resource partitions according to load estimates improves the slowdown of both Grid jobs (Figure 4a) and providers' local jobs but not as much as that of the other multiple partition policies.

We also vary the intervals for providing the free time slots in the previous experiment. Figure 5a shows that for small planning horizons, the multiple resource partition policy with EASY backfilling and load estimates (i.e. ArEbMf+earliest-partial) improves the average ratio, but not as much as the other policies. However, as the time interval for providing the availability information increases, the policy outperforms the other multiple partition policies. The slowdown improves compared to the other policies when the interval increases. The reason for the better performance under long intervals is probably because if a load estimate is wrong, the policy becomes a multiple partition conservative backfilling. When an incorrect estimate is identified in a long interval, it may take a while to approach the

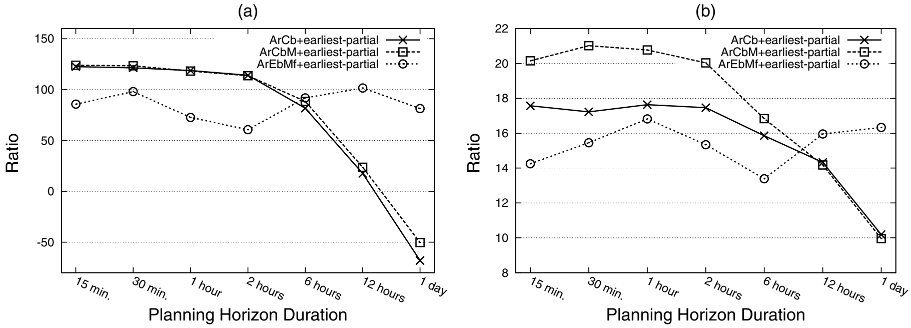


Fig. 5. Slowdown improvement ratio: (a) Grid jobs and (b) providers' local jobs

next interval when the policy will become EASY backfilling again. This conservative backfilling provides a better slowdown and updating the availability in the middle of an interval provides an advantage over the other policies. However, to confirm that, we require further investigation. Furthermore, we expect that better load forecast methods can improve the jobs slowdown under varying intervals.

6 Conclusions and Future Work

This work investigates resource provisioning in multiple site environments. It evaluates whether it is possible to provision resources for Grid applications based on availability information given by resource providers using existing resource management systems. We present empirical results that demonstrate that in an environment like DAS-2, a gateway can provision resources to Grid applications if the resource providers inform the available time slots between 15 and 30 minutes. Additionally, multiple resource partition policies can improve the slowdown of both local and Grid jobs if conservative backfilling is used. Future investigations include more sophisticated resource provisioning policies for the gateways and more accurate load forecasting techniques.

Acknowledgements

We thank the Grid Workloads Archive group for making the Grid workload traces available. This work is supported by DEST and ARC project grants. Marcos' PhD research is partially supported by National ICT Australia (NICTA).

References

1. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the Grid: Enabling scalable virtual organizations. Int. J. High Perform. Comput. Appl. 15(3), 200–222 (2001)
2. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Transactions on Parallel and Distributed Systems 12(6), 529–543 (2001)

3. Singh, G., Kesselman, C., Deelman, E.: A provisioning model and its comparison with best-effort for performance-cost optimization in Grids. In: 16th IEEE HPDC, Monterey, USA, pp. 117–126 (2007)
4. Lifka, D.A.: The anl/ibm sp scheduling system. In: Workshop on Job Scheduling Strategies for Parallel Processing (IPPS 1995), London, UK, pp. 295–303 (1995)
5. AuYoung, A., Grit, L., Wiener, J., Wilkes, J.: Service contracts and aggregate utility functions. In: 15th IEEE HPDC, Paris, France, pp. 119–131 (2006)
6. Röblitz, T., Schintke, F., Wendler, J.: Elastic Grid reservations with user-defined optimization policies. In: Workshop on Adaptive Grid Middleware, France (2004)
7. Wieczorek, M., Siddiqui, M., Villazon, A., Prodan, R., Fahringer, T.: Applying advance reservation to increase predictability of workflow execution on the Grid. In: 2nd IEEE e-Science, Washington DC, USA, p. 82 (2006)
8. Lawson, B.G., Smirni, E.: Multiple-queue backfilling scheduling with priorities and reservations for parallel systems. In: 8th JSSPP, London, UK, pp. 72–87 (2002)
9. Padala, P., Shin, K.G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., Salem, K.: Adaptive control of virtualized resources in utility computing environments. In: 2007 Conference on EuroSys, Lisbon, Portugal, pp. 289–302 (2007)
10. Garbacki, P., Naik, V.K.: Efficient resource virtualization and sharing strategies for heterogeneous Grid environments. In: 10th IFIP/IEEE IM, Munich, Germany, pp. 40–49 (2007)
11. Singh, G., Kesselman, C., Deelman, E.: Application-level resource provisioning on the grid. In: 2nd IEEE e-Science, Amsterdam, The Netherlands, p. 83 (2006)
12. Fu, Y., Chase, J., Chun, B., Schwab, S., Vahdat, A.: SHARP: An architecture for secure resource peering. In: 19th ACM Symposium on Operating Systems Principles, New York, NY, USA, pp. 133–148 (2003)
13. de Assunção, M.D., Buyya, R., Venugopal, S.: InterGrid: A case for internetworking islands of Grids. Conc. and Comp.: Pr. and Exp (CCPE) 20(8), 997–1024 (2008)
14. Jackson, D.B., Snell, Q., Clement, M.J.: Core algorithms of the Maui scheduler. In: 7th JSSPP, London, UK, pp. 87–102 (2001)
15. The Distributed ASCI Supercomputer 2: Dutch University Backbone (2006)
16. Iosup, A., Epema, D.H.J., Tannenbaum, T., Farrellee, M., Livny, M.: Interoperating Grids through delegated matchmaking. In: 2007 ACM/IEEE Conference on Supercomputing, Reno, USA (2007)
17. Lublin, U., Feitelson, D.G.: The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. J. Par. and Dist. Comp. 63(11), 1105–1122 (2003)
18. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., Sevcik, K.C., Wong, P.: Theory and practice in parallel job scheduling. In: Job Scheduling Strategies for Parallel Processing, London, UK, pp. 1–34. Springer, Heidelberg (1997)
19. Islam, M., Balaji, P., Sadayappan, P., Panda, D.K.: QoPS: A QoS based scheme for parallel job scheduling. In: 9th JSSPP, Seattle, USA, pp. 252–268 (2003)
20. Hanke, J.E., Reitsch, A.G.: Business Forecasting, 5th edn. Prentice-Hall, Inc., Englewood Cliffs (1995)

An Open Computing Resource Management Framework for Real-Time Computing

Vuk Marojevic, Xavier Revés, and Antoni Gelonch

Dept. of Signal Theory and Communications, Universitat Politècnica de Catalunya,
C/ Jordi Girona 1-3, Modul D4, 08034 Barcelona, Spain
{marojevic,xavier.reves,antoni}@tsc.upc.edu

Abstract. This paper introduces an open computing resource management framework for real-time computing systems. The framework is modular and consists of a general computing resource modeling that facilitates a policy-based (open) computing resource management. The computing resource modeling contains two resource model templates, which may be instantiated as often as necessary to capture a platform's computing resources and an application's computing requirements. The computing resource management approach features a parametric algorithm (t_w-mapping with window size w) and a generic and parametric cost function, which implements the computing resource management policy. We present simulations using a simple instance of this cost function to demonstrate the suitability and versatility of the framework. We compute a metric that relates the computing resource management success to its complexity and conclude that adjusting the cost function's parameter is more efficient than augmenting the t_w-mapping's window size.

Keywords: computing resource management, real-time computing, open framework.

1 Introduction

Many applications require huge amounts of computing resources. Multimedia applications or mobile communications systems, for example, need high processing powers for real-time data processing. Moreover, applications are often personalized for a particular user or user group, which demands more and more sophisticated services. This includes communication services but also other types of popular services, such as videostreaming. The solution to these computing demands is *multiprocessing*, where applications are processed in parallel on arrays of processors that offer much higher processing powers than single-processor execution environments. A single application can, generally, be parallelized and may then be executed together with other applications. In software-defined radio (SDR) [1], for example, a single-user mobile terminal would normally execute only a few applications, the radio and user applications in the most basic case, whereas a multi-user base station serves many users at a time and thus executes many applications concurrently.

Parallel or multiprocessing is more complicated than sequential processing because the available resources need to be shared spatially and temporally. A single processor

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 169–182, 2008.

executes applications sequentially, where pseudo-parallelism is achieved through assigning processing time slots to different applications or application's parts. Multiprocessor execution environments, on the other hand, allow for distributed computing, enabling true parallelism. The distributed resources though require an appropriate computing resource management. Computing resources include processing powers, inter-processor bandwidths, memory, and energy resources; in general, all those resources that are required for the execution of applications. Their time management is necessary for a real-time execution.

This paper introduces a new approach to real-time computing resource management. It presents a general framework that can efficiently manage the limited computing resources of multiprocessor platforms while providing the necessary amount of resources to real-time applications. The framework is not optimized for a specific objective but rather open to different management policies. We call this an *open* computing resource management framework. It is more general than our earlier proposal [2].

This framework bases itself on previous research results (section 2). It is a modular design that consists of two principal modules: the computing system modeling (section 3) and the computing resource management (section 4). The letter is further divided into the mapping algorithm (section 4.1) and the cost function (section 4.2). This modular design, in particular, the independence between the algorithm and its objective (cost function), facilitates exchanging the computing resource management policy. Numerous simulations demonstrate the versatility and suitability of the entire framework (section 5), leading to interesting conclusions that pave the path for future research (section 6).

2 Related Work

This work focuses on real-time computing systems. It particularly addresses real-time capable execution environments of limited computing resources and applications with real-time processing demands. We assume that the system's constraints—the applications' real-time computing requirements and the platforms' limited computing resources—have just to be met. Additional or other objectives, such as speeding up an application (more than strictly necessary to meet the given timing constraints), are thus irrelevant here. The framework accounts for platforms and applications with heterogeneous computing resources and requirements (*heterogeneous computing*).

Related work considers almost any problem and objective in heterogeneous computing. A vast amount of literature particularly addresses the mapping of real-time applications to multiprocessor platforms and the scheduling of processes and data flows. We consider *mapping* and *scheduling* as two complementary computing resource management methods and try to generalize previous efforts in heterogeneous computing, taking advantage of their results and conclusions. Due to space limitations, the following paragraphs detail only a few related contributions.

References [3] and [4] address the problem of optimally allocating periodic tasks, which are subject to task precedence and timing constraints, to processing nodes of a distributed real-time system. The efficient local scheduling of tasks in a real-time multiprocessor system is the topic of [5]. If a task's deadline cannot be met on a particular processing node, this task can be sent to another node [6]. The task model, which is identical in both papers, accounts for worst case computation times, deadlines, and resource requirements; no precedence constraints are assumed.

Instead of assuming worst-case application requirements, [7] proposes to adapt the resource allocation to face the runtime changes in the application environment. It describes and evaluates models and mechanisms for adaptive resource allocation in the context of embedded high performance applications with real-time constraints. Based on the same principle, [8] presents a mathematical modeling for an adaptive resource management in dynamic application environments. It precisely models fixed hardware—a network of processors—and dynamic, real-time software at different abstraction layers. It also proposes a framework for allocation algorithms, supporting the three constraints application-host validity, minimum security level, and real-time deadlines, while maximizing the overall utility of the system. Security is a common issue in recent publications, such as [9] which allocates computing resources to real-time and security-constrained parallel jobs.

A list scheduling framework for the run-time stabilization of static and dynamic tasks with hard and soft deadlines, respectively, is described in [10]. It allows for dynamic or static task-to-processor allocations and implements mechanisms that control the degree of resource reclaiming to increase the processor utilization and the response time of the dynamic workload. Reference [11] tackles hard real-time streaming applications in a scenario where jobs enter and leave a particular homogeneous multiprocessor system at any time during operation. It combines global resource allocation (mapping) with local resource provisioning (scheduling).

Other related contributions are [12]-[14]. Although they do not specifically address real-time systems, they deal with particular aspects that this framework adopts. The dynamic level scheduling (DLS) approach [12], for example, accounts for interprocessor communication overheads. It maps precedence-constrained, communicating tasks to heterogeneous processor architectures with limited or irregular interconnection structures. Alhusaini et al. introduce the problem of resource co-allocation, which refers to simultaneous allocations of different types of resources that are shared among applications, and formulate the mapping problem in the presence of co-allocation requirements [13]. Reference [14], finally, introduces a theory for scheduling directed acyclic graphs (DAGs) in internet-based computing. It applies graph theory techniques to model precedence-constrained computing tasks and to derive optimal schedules for different types of DAGs.

3 Computing System Modeling

This section presents a mathematical modeling of computing resources (section 3.1) and requirements (section 3.2). It features resource model templates that can be instantiated as many times as necessary to capture the relevant resources and requirements. We assume the availability of a middleware or hardware abstraction layer, such as [15], that facilitates the information about hardware capacities and software requirements in the units specified below.

3.1 Computing Resources

$\boldsymbol{R}^t \in \mathbb{R}^+ \times \mathbb{R}^+$ represents the template for modeling the computing environment. ($\mathbb{R}^+$ symbolizes non-negative real numbers.) It is an $X(t)$ times $Y(t)$ matrix ($X(t)$, $Y(t) \in \mathbb{N}$),

$$\boldsymbol{R}^t = \begin{pmatrix} R^t_{11} & R^t_{12} & \cdots & R^t_{1,Y(t)} \\ R^t_{21} & R^t_{22} & \cdots & R^t_{2,Y(t)} \\ \vdots & \vdots & \ddots & \vdots \\ R^t_{X(t),1} & R^t_{X(t),2} & \cdots & R^t_{X(t),Y(t)} \end{pmatrix}, \tag{1}$$

where $t \in 1, 2, \ldots, T$ denotes the resource model index and T the number of $\boldsymbol{R}^t$ instances. $\boldsymbol{R}^t$ is apt for characterizing different types of computing architectures and capturing the available computing resources, such as processing powers and interprocessor bandwidths. It is therefore unitless.

$$\boldsymbol{R}^1 = \boldsymbol{C} = [C_1, C_2, \ldots, C_N] \text{ MOPS} \tag{2}$$

models the processing powers of processors P_1, P_2, …, P_N in million operations per second (MOPS). It instantiates (1) with $X(1) = 1$, $Y(1) = N$, and $R^1_{1i} = C_i$. Without loss of generality, we label devices in order of decreasing processing capacities, that is, $C_1 \geq C_2 \geq \ldots \geq C_N$.

$$\boldsymbol{R}^2 = \boldsymbol{I} = \begin{pmatrix} I_{11} & I_{12} & \cdots & I_{1N} \\ I_{21} & I_{22} & \cdots & I_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ I_{N1} & I_{N2} & \cdots & I_{NN} \end{pmatrix} = \begin{pmatrix} 1 & I_{12} & \cdots & I_{1N} \\ I_{21} & 1 & \cdots & I_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ I_{N1} & I_{N2} & \cdots & 1 \end{pmatrix} \tag{3}$$

represents the logical interconnection model. A logical link corresponds to a directed (unidirectional) communication line between a pair of processor. These logical links map to physical links: A logical link between any two processors maps to a physical link of a certain bandwidth if the two processors are actually connected; otherwise it maps to an imaginary physical link of zero bandwidth. Mathematically, $\boldsymbol{I}$ holds the indexes that point to the physical link bandwidths, which we model as

$$\boldsymbol{R}^3 = \boldsymbol{B} = [B_1, B_2, B_3, \ldots, B_{N\cdot[N-1]+1}] = [\infty, B_2, B_3, \ldots, B_{N\cdot[N-1]+1}] \text{ MBPS.} \tag{4}$$

B_x, where $x = I_{32}$ for instance, is the maximum bandwidth in mega-bits per second (MBPS) that is available for the directed data transfer from the local data memory of processor P_3 to the local data memory of processor P_2. B_1 models the processor-internal bandwidth capacities, assuming direct memory access (DMA) or pointer transfers for processor-internal data flows.

Equations (3) and (4) facilitate modeling shared or bidirectional buses, mapping the corresponding logical links to a single entry in $\boldsymbol{B}$. Unnecessary positions in (4) are then filled with zeros. $\boldsymbol{I}$ can be organized in such a way that $B_2 \geq B_3 \geq \ldots \geq B_{N\cdot[N-1]+1}$.

This section has presented three instances of (1); one captures the communication architecture ($\boldsymbol{I}$) and two actual computing resources ($\boldsymbol{C}$ and $\boldsymbol{B}$). We introduce $t' \in 1, 2, \ldots, T'$, which indexes the instances of (1) that are actual computing resources. Then,

$t' = 1$ and $t' = 2$ index $\boldsymbol{C}$ and $\boldsymbol{B}$. Modeling the actual computing resources per time unit facilitates handling real-time applications with limited resources (section 4).

3.2 Computing Requirements

Matrix $\boldsymbol{r}^t \in \mathbb{R}^+ \times \mathbb{R}^+$ is the applications' general computing model. It is, equivalently to $\boldsymbol{R}^t$, an $x(t)$ times $y(t)$ matrix ($x(t)$, $y(t) \in \mathbb{N}$):

$$\boldsymbol{r}^t = \begin{pmatrix} r_{11}^t & r_{12}^t & \cdots & r_{1,y(t)}^t \\ r_{21}^t & r_{22}^t & \cdots & r_{2,y(t)}^t \\ \vdots & \vdots & \ddots & \vdots \\ r_{x(t),1}^t & r_{x(t),2}^t & \cdots & r_{x(t),y(t)}^t \end{pmatrix}. \tag{5}$$

We model an application as M processes that process and propagate data. Then we can introduce instances of (5) that correspond to instances (2)–(4) of (1).

$$\boldsymbol{r}^1 = \boldsymbol{c} = [c_1, c_2, \ldots, c_M] \text{ MOPS}, \tag{6}$$

particularly, resumes the processing requirements of processes p_1 to p_M. We assume that applications' processing chains represent directed acyclic graphs (DAGs) [12]-[14]; cyclic dependencies are then process-internal. DAGs can be logically numbered: If p_x sends data to p_y, then $y > x$ [16]. This leads to

$$\boldsymbol{r}^2 = \boldsymbol{i} = \begin{pmatrix} i_{11} & i_{12} & \cdots & i_{1M} \\ i_{21} & i_{22} & \cdots & i_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ i_{M1} & i_{M2} & \cdots & i_{MM} \end{pmatrix} = \begin{pmatrix} 1 & i_{12} & \cdots & i_{1M} \\ 1 & 1 & \cdots & i_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{pmatrix} \tag{7}$$

and

$$\boldsymbol{r}^3 = \boldsymbol{b} = [b_1, b_2, b_3, \ldots, b_{M\cdot[M-1]/2+1}] = [0, b_2, b_3, \ldots, b_{M\cdot[M-1]/2+1}] \text{ MBPS}. \tag{8}$$

Equation (7) indicates an application's precedence constraints and together with (8) represents the dataflow requirements: b_x, where $x = i_{12}$, for instance, is the minimum bandwidth that is necessary to transmit data from process p_1 to process p_2 in real time. Unreferenced elements in $\boldsymbol{b}$ are filled with zeros. Organizing the upper diagonal of $\boldsymbol{i}$ so that $b_2 \geq b_3 \geq \ldots \geq b_{M\cdot[M-1]/2+1}$ facilitates implementing certain mapping techniques.

Separating the precedence constraints from the bandwidth demands facilitates distinguishing between dependent and independent data flows. For example, if p_1 sends the same data to p_3 and p_4, i_{13} and i_{14} may point to the same entry in $\boldsymbol{b}$ ($i_{13} = i_{14}$ – dependent data flows), whereas if p_1 sends two different data chunks, one to p_3 and one to p_4, i_{13} and i_{14} should point to the different entries in $\boldsymbol{b}$ ($i_{13} \neq i_{14}$ – independent data flows). This paper considers independent data flows.

The processing or bandwidth requirements can be obtained from multiplying the number of operations or bits that need to be processed or propagated by the available time for doing so. This correctly models applications' real-time requirements.

4 Computing Resource Management

4.1 The t_w-Mapping

The t_w-mapping was introduced in [2]. Here we summarize its principal characteristics. It is a windowed dynamic programming algorithm, where w indicates the window size. It is organized through the t_w-mapping diagram (Fig. 1), which contains a trellis of N x M (row x column) t-nodes. A t-node is identified as $\{P_{k(l)}, p_s\}$ and absorbs the mapping of process p_s to processor $P_{k(l)}$. Any t-node at *step* s (column s in the t_w-mapping diagram) connects to all t-nodes at *step* s+1. The sequence of processors $[P_{k(0)}\ P_{k(1)}\ ...\ P_{k(w)}]_s$ identifies the w-path, a path of length w, that is associated with $\{P_{k(1)}, p_s\}$, where $P_{k(0)}$ is the w-path's origin processor at *step* s–1 and $P_{k(w)}$ the destination processor at *step* s+w–1. Table 1 contains the most important variables and expressions that appear in the rest of the paper.

The main feature of the t_w-mapping is that it is cost function independent. That is, any cost function can, in principle, be applied. The cost function guides the mapping process. It is responsible for managing a platform's available computing resources and an application's real-time processing requirements (section 4.2).

The algorithm sequentially pre-assigns, or pre-maps, processes to processors, starting with process p_1 and finishing with process p_M (parts I and II of the t_w-mapping). This is followed by a post processing that determines the final mapping (part III).

Table 1. Ranges and descriptions of variables and expressions

Variable or expression	Range (Argument range)	Description
N	1, 2, …	number of processors
M	1, 2, …	number of processes
w	1, 2, …, M–1	window size
$k(l)$	1, 2, …, N; ($l \in$ 0, 1, …, w)	processor index $k(l)$ with its relative position l in the w-path
$P_{k(l)}$	$P_1, P_2, …, P_N$	processor
s	1, 2, …, M	step index (process index)
p_s	$p_1, p_2, …, p_M$	process
$\{P_{k(l)}, p_s\}$		t-node indicating the mapping of p_s to $P_{k(l)}$
$[P_{k(0)}\ P_{k(1)}\ ...\ P_{k(w)}]_s$	($s \in$ 2, 3, …, M–w+1)	w-path associated with $\{P_{k(1)}, p_s\}$
$h = s + (l – 1)$	($l \neq 0$; $s \in$ 2, 3, …, M–w+1)	step index h substitutes $s + (l – 1)$
$[P_{k(l–1)}\ P_{k(l)}]_h$	($l \neq 0$; $s \in$ 2, 3, …, M–w+1)	edge between $\{P_{k(l–1)}, p_{h–1}\}$ and $\{P_{k(l)}, p_h\}$
WT$[P_{k(l–1)}\ P_{k(l)}]_h$	($l \neq 0$; $s \in$ 2, 3, …, M–w+1)	edge weight
$\boldsymbol{R}^{t'\,@\{k(l),\,h\}}$		remaining computing resources of type t' at $\{P_{k(l)}, p_h\}$
$\boldsymbol{r}^{t'\,@\{k(l),\,h\}}$		required computing resources of type t' at $\{P_{k(l)}, p_h\}$

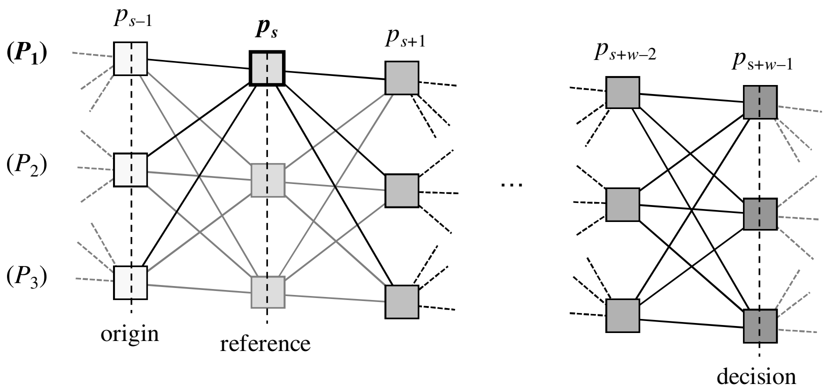


Fig. 1. Extract of the t_w-mapping diagram for three processors ($N = 3$). The black edges indicate those w-paths that are examined at t-node $\{P_1, p_s\}$, assuming $w \geq 3$.

Part I consists of pre-mapping process p_1 to all N processors and storing the pre-mapping costs at t-nodes $\{P_1, p_1\}$ through $\{P_N, p_1\}$. Costs are computed due to some cost function.

At step s of part II ($2 \leq s \leq M–w+1$) the t_w-mapping examines all N^w w-paths that are associated with $\{P_{k(1)}, p_s\}$. These w-paths originate at a t-node at step s–1, pass through $\{P_{k(1)}, p_s\}$, and terminate at a t-node at step s+w–1. Fig. 1 illustrates this for $P_{k(1)} = P_1$.

In case that $s < M–w+1$, the algorithm highlights the edge between a t-node at step s–1 and t-node $\{P_{k(1)}, p_s\}$ that corresponds to the minimum-cost w-path. The minimum-cost w-path is the path that is associated with the minimum accumulated cost due to the corresponding pre-mappings of p_1, p_2, …, and p_{s+w-1}, where the w-path's origin t-node provides the pre-mapping information of p_1 to p_{s-1}. The algorithm then stores the cost and the remaining resources up to t-node $\{P_{k(1)}, p_s\}$ at $\{P_{k(1)}, p_s\}$. It (simultaneously) processes all t-nodes at step s before considering those at step s+1.

If $s = M–w+1$, however, the entire minimum-cost w-path is highlighted. The total cost and finally remaining resources are then stored at $\{P_{k(1)}, p_{M-w+1}\}$. After having processed all N t-nodes at step M–w+1, part III of the algorithm follows.

Part III tracks the t_w-mapping diagram back- and forward along the highlighted edges, starting at the t-node at step M–w+1 that holds the minimum cost. This process finds the complete mapping solution for the given problem and cost function.

The algorithm's complexity depends on the cost function. Assuming that the complexity of the cost function (*ccf*) is constant throughout the mapping process, we can write

$$\text{complexity}(t_w\text{-mapping}) \approx (M - w) \cdot N^2 \cdot \frac{N^w - 1}{N - 1} \cdot ccf\,. \tag{9}$$

Considering that $M >> w$ and $ccf = 1$, the order of magnitude becomes

$$\text{complexity-order}(t_w\text{-mapping}) = O(M \cdot N^{w+1}). \tag{10}$$

This indicates that the algorithm is not computing efficient for large N. We therefore suggest to cluster (huge) arrays of processors, which will eventually execute many applications, and to apply the t_w-mapping on each cluster.

4.2 Cost Function

This section proposes a generic cost function that manages the available computing resources based on our modeling concept. We define it through the edge weight:

$$\mathrm{WT}[P_{k(l-1)}\ \ P_{k(l)}]_h = \sum_{t'=1}^{T'} q_{t'} \cdot cost_{t'}^{@\{k(l),h\}}, \tag{11a}$$

where

$$cost_{t'}^{@\{k(l),h\}} = \begin{cases} \sum\limits_{i,j} \dfrac{r_{ij}^{t'@\{k(l),h\}}}{R_{i'j'}^{t'@\{k(l),h\}}}, & \text{if } \dfrac{r_{ij}^{t'@\{k(l),h\}}}{R_{i'j'}^{t'@\{k(l),h\}}} \le 1 \ \forall i,j \\ \infty, & \text{otherwise} \end{cases}. \tag{11b}$$

$[P_{k(l-1)}\ P_{k(l)}]_h$ represents any edge in the t_w-mapping diagram; it is for $w > 1$ part of a w-path. Each summand in (11a) stands for the weighted cost for the allocations of resource type t' at t-node $\{P_{k(l)}, p_h\}$. Equation (11b) defines this cost as the sum of ratios between the required resources ($r_{ij}^{t'}$) and the corresponding remaining ones ($R_{i'j'}^{t'}$) at $\{P_{k(l)}, p_h\}$. This implies a dynamic resource update.

Each ratio between a resource requirement and its availability is either less, equal, or greater than 1. In the latter case, it is mapped to infinity (11b). Hence, assuming $q_{t'} \neq 0 \, \forall \, t'$, the weighted sum in (11a) returns either a finite or infinite value. A finite edge weight indicates a feasible, an infinite an infeasible pre-mapping. This permits identifying and discarding infeasible solutions, which cannot meet the system's real-time computing constraints.

Cost function (11) defines the computing resource management policy through parameter $\boldsymbol{q}$, where $\boldsymbol{q} = [q_1, q_2, \ldots, q_{T'}]$ weights the cost terms. The higher $q_{t'}$ the higher the relative importance of resource type t'. The sum of all weights can be normalized to 1. Then, $q_1 = q_2 = \ldots = q_{T'} = 1/T'$ would mean equally weighted cost terms.

4.3 Scheduling

On the basis of a feasible mapping—a mapping of finite cost—N processor-local schedulers need to schedule processes, data transfers, and possibly other resource allocations to guarantee that real-time constraints will finally be met. Finding such schedules is possible if we assume that processing chains can be pipelined, that data processing and data transfers can overlap, and that partial results can be immediately forwarded to the next process [2].

Access to any shared resource requires its temporal management or scheduling. Each shared link, for example, requires a scheduler. Assuming the availability of data buffers, these schedulers can use a simple policy to ensure timely data transfers: Transfer data immediately to output data buffers. This data is sent to the corresponding input buffer as soon as the bus becomes available, gaining access to the different processors that share the bus in a round-robin fashion.

5 Simulations

5.1 Simulation Setup

The following simulations serve for demonstrating the suitability and possibilities of the framework. Due to space limitations we consider a single computing platform and two computing resources ($T' = 2$)—processing powers ($t' = 1$) and inter-processor bandwidths ($t' = 2$)—based on the system model instances of section 3. Fig. 2 shows the computing platform model. This platform may represent a stand-alone computing cluster of three heterogeneous devices or be connected to an array of processors.

We randomly generate 100,000 DAGs, which model different applications, where

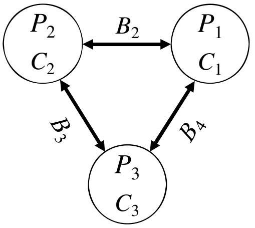
- the number of processes is $M = 25$,
- process p_i is connected to p_j with a probability of 0.2 if $j > i$ (we allow disconnected subgraphs, modeling parallel chains, but connect any isolated node to its next neighbor),
- the processing demands are uniformly distributed in [1, 2, ..., 600] MOPS, and
- the bandwidth demands are uniformly distributed in [1, 2, ..., 500] MBPS.

These parameters have been derived from a real SDR application [2]. Nevertheless, this random DAG generation results in many different application topologies (precedence constraints) and computing requirements. The mean processing requirement is 7429.8 MOPS, which is slightly less than $25 \cdot (600+1)/2 = 7512.5$ MOPS, because we discard applications which need more than the 9000 available MOPS (Fig. 2). 7429.8 MOPS correspond to 82.6 % of the platform's total processing resources. An application's total bandwidth requirement is 15,069.2 MBPS in the mean, being 167.4 % of the available inter-processor communication bandwidths.

5.2 Results I: Ordering

The t_w-mapping with cost function (11) maps computing requirements to computing resources. It does so sequentially, starting with process p_1 and finishing with process p_M (section 4). Related work demonstrated the importance of the mapping or scheduling order. Reference [12], for instance, assigns dynamic levels to determine the next process to be scheduled.

The modeling of section 3.2 facilitates the reordering or relabeling of processes through basic matrix operations. In particular, to change process p_i for p_j, exchange c_i and c_j in (6) and switch rows and columns i and j in (7). Switching rows i and j in (7)



$$\boldsymbol{C} = (4000, 3000, 2000) \text{ MOPS}$$

$$\boldsymbol{I} = \begin{pmatrix} 1 & 2 & 4 \\ 2 & 1 & 3 \\ 4 & 3 & 1 \end{pmatrix}$$

$$\boldsymbol{B} = (\infty, 3000, 3000, 3000, 0, 0, 0) \text{ MBPS}$$

Fig. 2. Computing platform model

changes the successors of p_i for those of p_j and vice versa, whereas switching columns i and j changes the predecessors of p_i for those of p_j and vice versa. This maintains the same DAG with another labeling of processes.

Here we study static reordering techniques. Dynamic reordering of the remaining processes to be mapped will be examined in future work. We consider 3 approaches:

- no reordering (ORD-0),
- reordering by decreasing processing requirements (ORD-C), and
- reordering by decreasing bandwidth demands (ORD-B).

ORD-0 assumes the initial order based on a logical numbering, which is generally not unique. ORD-C leads to $c_1 \geq c_2 \geq \ldots \geq c_M$, whereas in case of ORD-B, the pair of processes with the heaviest data flow demand become p_1 and p_2, the next highest bandwidth requirement specifies p_3 (and p_4), and so forth. The flexibility of mapping lower computing requirements to the remaining computing resources is the reason for choosing ORD-C or ORD-B. Simulations will show which approach is more suitable for the considered scenario.

Table 2 shows the t_w-mapping results for three values of q_1. We observe that ORD-B and ORD-C outperform ORD-0. This can be explained as follows: Bandwidth resources, as opposed to processing resource, can be saved if (heavily) communicating processes are mapped to the same processor. (This is why we can map applications that have a higher total bandwidth requirement than the platform's total inter-processor bandwidth capacity.) Saving bandwidths is only possible if there is a processor with sufficient processing capacities for executing two communicating processes. If heavily communicating processes are considered first (ORD-B), it is most probable that heavy links can be solved processor-internally. Correspondingly, the flexibility of mapping lower processing requirements can potentially merge heavily communicating processes and explains the good behavior of ORD-C.

Additional simulations have shown that ORD-C is more suitable than ORD-B if the processing resources are the bottleneck, whereas ORD-B performs better than ORD-C if the bandwidth requirements are dominating. Here, the high processing and bandwidth loads (section 5.1) explain the similar performances of ORD-C and ORD-B. Since the best results are obtained for q_1 = 0.7 and ORD-B (Table 2), the following simulations apply the ORD-B algorithm before executing the t_w-mapping.

Table 2. Results I

	w = 1			w = 3		
q_1	ORD-0	ORD-C	ORD-B	ORD-0	ORD-C	ORD-B
0.3	33.73	15.39	16.12	19.06	7.39	8.95
0.5	25.07	11.48	11.31	13.61	6.05	6.05
0.7	21.97	13.09	9.92	11.71	6.58	5.13

5.3 Results II: Cost Function Parameter *q* vs. Window Size *w*

Methods. First we consider a fixed q vector for all 100,000 DAGs. We examine q_1 = 0.1, 0.2, …, 0.9 to obtain the optimal q in the mean (Method A). Then we propose to choose q_1 dynamically, trying different values until either a feasible mapping is found or all values have been examined. In particular, q_1 is iteratively updated in the following order: 0.5, 0.4, 0.6, 0.3, 0.7, 0.2, 0.8, 0.1, 0.9 (Method B). This is a simple method that considers q_1 at a granularity of 0.1, starting with equally weighted cost terms and discarding single-term cost functions. The number of iterations, or mapping intents per application, is then between 1 (for a successful mapping with $q_1 = 0.5$) and 9 (if $q_1 = 0.9$ is finally examined). The mean number of iterations (*m-iter*) is the number of iterations averaged over the considered DAGs.

Metric. In order to compare the two methods and to formalize the significance of the cost function parameter q versus the window size w, we introduce

$$\textit{metric-I} = \textit{quality} \,/\, \textit{complexity}. \tag{12}$$

This metric relates the quality of the computing resource management approach to its computing complexity. It may be considered an efficiency indicator, because efficiency indicates good results at little effort.

Here we define *quality* as follows: If the algorithm fails in mapping x % of the applications, its *quality* is $1/x$. We measure the complexity in two ways, theoretically and practically. In both cases we count the number of multiply-accumulate operations (MACs), where 1 MAC stands for one multiplication or division followed by a summation.

The theoretical complexity for the given two-term cost function can be approximated as

$$\textit{theoretical-complexity} \approx (M - w) \cdot N \cdot \left\{ \sum_{k=1}^{w} N^k \cdot \frac{M - w + 2k + 1}{2} + 2N^w \right\}. \tag{13}$$

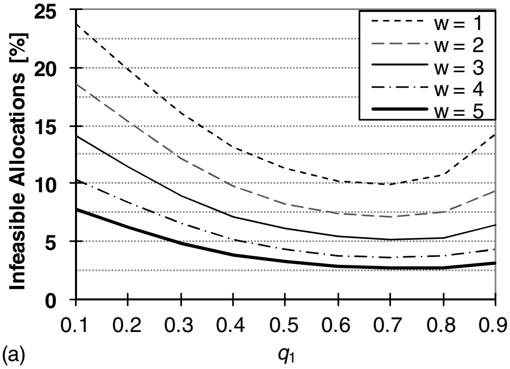
The different terms in (13) are:

- $(M - w)$ is the number of steps that pertain to part II of the t_w-mapping,
- N is the number of t-nodes per step, and
- $\{\cdot\}$ represents the complexity of computing the cost due to processing and bandwidth requirements at a t-node of part II, where $2N^w$ are the additional 2 MACs for multiplying each w-path's cost term with q_1 and q_2.

Equation (13) approximates the theoretical maximum complexity. It accounts for fully connected DAGs, dividing the bandwidth requirement of each possible link between processes by the corresponding finite or infinite bandwidth capacity.

The theoretical complexity for method B is computed as *m-iter* times (13). The practical complexity model accounts for code optimizations: The practical complexities are obtained from C-code implementations, counting each MAC that is actually realized.

Results. Fig. 3a shows the percentage of unfeasibly mapped DAGs due to method A as a function of q_1 and w. It clearly indicates that the mapping success is a function of the window size. We further observe that the lowest number of infeasible mappings is achieved with $q_1 = 0.7$ for any w. The number of infeasible allocations is two to three times lower with $q_1 = 0.7$ than it is with the least favorable q_1, which is $q_1 = 0.1$. This justifies Method B's order of examining the different q_1 values. (Another order would

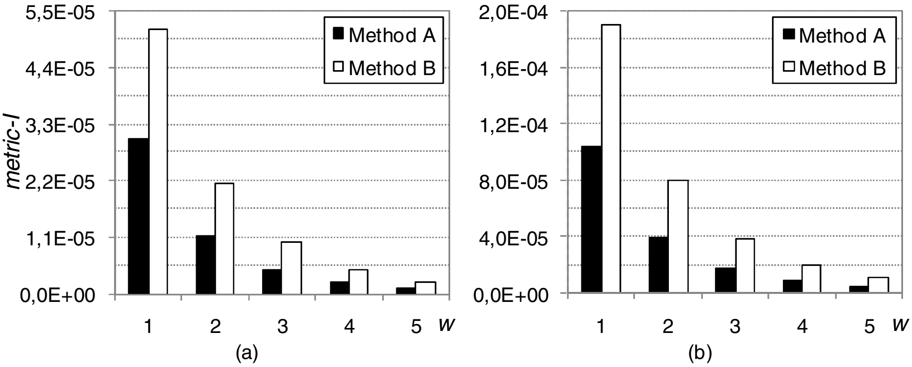


(a)

w	Infeasible allocations [%]	*m-iter*
1	3.79	1.53
2	2.64	1.38
3	1.94	1.28
4	1.42	1.21
5	0.98	1.15

(b)

Fig. 3. Results II with method A (a) and method B (b)



(a) (b)

Fig. 4. *metric-I* for methods A and B based on theoretical (a) and practical (b) complexities

affect *m-iter* but not the percentage of infeasible allocations.) Fig. 3b shows the corresponding results. We again observe that the higher *w* the fewer the number of infeasible mappings. The mean number of iterations decreases correspondingly. It is generally low because more than 87.5 % of the DAGs are successfully mapped for q_1 = 0.5 (Fig. 3a), that is, after 1 iteration.

Fig. 3 shows that choosing $\boldsymbol{q}$ dynamically significantly reduces the number of infeasible results. This leads to the conclusion that $\boldsymbol{q}$ should be chosen on application basis (Method B) but also reinforces its importance within cost function (11).

We compute *metric-I* to formally compare the two methods and to discuss the dynamic selection of $\boldsymbol{q}$ versus the increase of *w*. Fig. 4 illustrates the results as a function of *w* for both, the theoretical and practical complexities. The practical complexities and the qualities for method A are based on $\boldsymbol{q}$ = [0.7, 0.3].

We observe that *metric-I* based on theoretical complexity numbers (Fig. 4a) is qualitatively equivalent to *metric-I* for practical complexities (Fig. 4b). We interpret this as a validation of the theoretical and practical complexity models.

Fig. 4 shows that searching for a suitable $\boldsymbol{q}$ on application basis—even with the basic $\boldsymbol{q}$-selection algorithm of method B—approximately doubles the proposed metric for any *w*. This metric considerably decreases with *w*; adjusting $\boldsymbol{q}$ is thus more interesting than increasing w. In particular, if the quality of the t_w-mapping results for some w and $\boldsymbol{q}$ is insufficient, we can improve it trying other $\boldsymbol{q}$ vectors without sacrificing efficiency. For example, a mapping success of 95 % is achieved with $w = 1$ in case of method B (Fig. 3b), whereas method A needs at least a window size of 3 (Fig. 3a). Relating the corresponding values of *metric-I*, we obtain a difference of one order of magnitude in favor of method B (Fig. 4). On the other hand, we argue for a complementary adjustment of both parameters. For instance, to achieve less than 3 % infeasible allocations, apply method B with $w = 2$ (instead of method A with $w = 5$).

The above conclusions demonstrate the suitability of the two parameters w and $\boldsymbol{q}$ but also validate *metric-I*. These conclusions are valid for the above simulations. Other problems may behave differently and so may require similar simulations to derive corresponding conclusions and appropriate parameter adjustments.

6 Conclusions

This paper has introduced a computing resource management framework for real-time systems. It consists of a modular design, which features systematically extensible computing system models and an open computing resource management approach. This approach comprises the t_w-mapping—a cost function independent mapping algorithm—and a generic cost function, which manages the available computing resources of any type to satisfy the applications' real-time execution demands.

The simulations have demonstrated the suitability of the entire framework as well as the significance of the two independent parameters w and $\boldsymbol{q}$; the proper adjustment of these parameters can significantly enhance the efficiency of the computing resource management. There is still room for improvement: A low-complex algorithm that dynamically reorders the remaining processes to be pre-mapped or dynamic adjustments of parameters w and $\boldsymbol{q}$ throughout the t_w-mapping process may further increase *metric-I*. We will investigate these issues as well as simulate scenarios with additional computing resource types, such as memory and energy.

This work is focused on real-time computing systems, where the objective is to meet real-time execution demands with limited computing resources. We are currently examining how to adapt the framework to other types of systems and objectives.

Acknowledgments. This work was supported by the Spanish National Science Council CYCIT under Grant TEC2006-09109, which is partially financed from the European Community through the FEDER program.

References

1. Mitola, J.: The software radio architecture. IEEE Commun. Mag. 33(5), 26–38 (1995)
2. Marojevic, V., Revés, X., Gelonch, A.: A computing resource management framework for software-defined radios. In: IEEE Trans. Comput. 57(10), 1399–1412 (2008)
3. Peng, D.-T., Shin, K.G., Abdelzaher, T.F.: Assignment and scheduling communicating periodic tasks in distributed real-time systems. IEEE Trans. Software Eng. 23(12), 745–758 (1997)

4. Hou, C.-J., Shin, K.G.: Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. IEEE Trans. Comput. 46(12), 1338–1356 (1997)
5. Ramamritham, K., Stankovic, J.A., Shiah, P.-F.: Efficient scheduling algorithms for real-time multiprocessor systems. IEEE Trans. Parallel Distrib. Syst. 1(2), 184–194 (1990)
6. Ramamritham, K., Stankovic, J.A., Zhao, W.: Distributed scheduling of tasks with deadlines and resource requirements. IEEE Trans. Comput. 38(8), 1110–1123 (1989)
7. Rosu, D., Schwan, K., Yalamanchili, S., Jha, R.: On adaptive resource allocation for complex real-time applications. In: Proc. 18th IEEE Int'l. Real-Time Systems Symp., pp. 320–329 (1997)
8. Ecker, K., et al.: An optimization framework for dynamic, distributed real-time systems. In: Proc. 17th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS 2003) (2003)
9. Xie, T., Qin, X.: Security-aware resource allocation for real-time parallel jobs on homogeneous and heterogeneous clusters. IEEE Trans. Parallel Distrib. Syst. 19(5), 682–697 (2008)
10. Krings, A.W., Azadmanesh, M.H.: Resource reclaiming in hard real-time systems with static and dynamic workloads. In: Proc. 30th IEEE Hawaii Int'l. Conf. System Sciences (HICSS), pp. 116–625 (1997)
11. Moreira, O., Mol, J.-D., Beckooij, M., van Meerbergen, J.: Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In: Proc. 11th IEEE Real Time Embedded Technology and Applications Symp. (RTAS 2005), pp. 332–341 (2005)
12. Sih, G.C., Lee, E.A.: A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. IEEE Trans. Parallel Distrib. Syst. 4(2), 175–187 (1993)
13. Alhusaini, A.H., Prasanna, V.K., Raghavendra, C.S.: A framework for mapping with resource co-allocation in heterogeneous computing systems. In: Proc. 9th Heterogeneous Computing Workshop (HCW 2000), pp. 273–286. IEEE CS Press, Los Alamitos (2000)
14. Malewicz, G., Rosneberg, A.L., Yurkewych, M.: Toward a theory for scheduling DAGs in internet-based computing. IEEE Trans. Comput. 55(6), 757–768 (2006)
15. Revés, X., Gelonch, A., Marojevic, V., Ferrus, R.: Software radios: unifying the reconfiguration process over heterogeneous platforms. EURASIP J. Applied Signal Processing 2005(16), 2626–2640 (2005)
16. Robinson, D.F., Foulds, L.R.: Digraphs: Theory and Techniques. Gordon and Breach Science Publisher Inc. (1980)

A Load Aware Channel Assignment and Link Scheduling Algorithm for Multi-channel Multi-radio Wireless Mesh Networks

Arun A. Kanagasabapathy, A. Antony Franklin, and C. Siva Ram Murthy

Indian Institute of Technology Madras
{akarun,antony}@cse.iitm.ac.in, murthy@iitm.ac.in

Abstract. Wireless Mesh Networks with multiple channels and multiple radios on the mesh routers, have a great potential to increase the overall capacity of the network. To obtain the complete benefit of the increased capacity of such networks, efficient Channel Assignment (CA) and Link Scheduling (LS) algorithms are extremely important. A static CA and LS may not be optimal, in terms of utilization of underlying network resources, for every traffic demand in the network. In this paper, we find the optimal CA and LS for the given traffic demand by formulating an MILP by considering the traffic demand for each source-destination pairs, with the objective of maximizing the total achieved throughput of the network. We also propose a simple, but effective Load aware Channel Assignment and Link Scheduling (LoCALS), a polynomially bound heuristic algorithm for CA and LS. We show that LoCALS performs on par with the optimal solution. Finally, we compare LoCALS with a distributed channel assignment algorithm which is unaware of the traffic demand, in order to demonstrate the importance of considering the traffic demand for CA and LS.

1 Introduction

Wireless Mesh Networks (WMNs) have become the most cost-effective option for wide scale deployment in the last mile wireless networks. WMNs consist of two kinds of network elements namely mesh routers and mesh clients. The mesh routers form the backbone network and the mesh clients are users of the WMN that generate traffic in the network. The mesh routers with the gateway functionality provide easy integration with wired networks and in particular provide connectivity to the Internet to the mesh clients. Several commercial deployments of WMN are already operational in the real world.

The capacity of a WMN can be increased by using multiple orthogonal (non-overlapping) channels for transmission and thereby improving the *channel spatial reuse*. To tap the complete potential of the multiple channels, the mesh routers must be equipped with multiple radios. Raniwala *et al.* [1] showed that there is a non-linear increase in capacity with the increase in number of radios in a Multiple Channel - Multiple Radio (MC-MR) networks. Though there is a potential increase in capacity due to the usage of MC-MR, a poor Channel Assignment

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 183–195, 2008.

(CA) scheme can lead to under-utilizing the network's capabilities. Intelligent CA schemes need to be adopted to spatially separate the nodes transmitting in the same channel as much as possible. Nevertheless, owing to the constraint on the number of channels and the number of radios available at each router, the co-channel interference can not be avoided completely. A network, employing carrier sensing techniques like CSMA/CA for transmission of data, consumes more power and loses bandwidth due to interference in the network. Also the carrier remains idle during the back-off time after every collision. The static nature of the mesh routers can be utilized to provide link level synchronization among the mesh routers. Efficient Link Scheduling (LS) can make the network interference free and thus improving the capacity of the network further.

There has been significant amount of work done in WMNs in recent years to exploit the usage of MC-MR in order to increase the achievable capacity of the network using new MAC and routing protocols. Bahl *et al.* [2] developed Slotted Seeded Channel Hopping (SSCH), a link layer solution for enhancing capacity of the network. Raniwala *et al.* [1] presented a centralized heuristic algorithm for CA and routing which iterates between routing and CA until the process converges in terms of the aggregate capacity of the network. The worst performance bound on their heuristic is not known. Tang *et al.* [3] proposed Linear Programming (LP) and Convex Programming (CP) based schemes for computing end-to-end fair rate allocation for WMNs. Their work heavily depends upon the ability of the transceivers to change the transmission power for every time slot, which may not be practically viable. Also Alicherry *et al.* [4] proposed a centralized joint CA, LS, and routing for WMNs. The objective is to increase the per node throughput in the network. The proposed heuristic algorithm is quite complex and spans multiple stages.

In this work, we design a CA and LS algorithm according to the traffic demand in the network. In a typical WMN, the traffic on the backbone network consists of the traffic between clients in the network and between clients and the Internet. This problem when modelled with the mesh routers alone (by not considering the mesh clients in the model), reduces to each node pair having certain amount of traffic to send between them. The assumption here is that the communication between the client and mesh router does not interfere with the communication between the mesh routers as they use different technologies for communication. The traffic demand between every node (mesh router) pair is known prior to the CA and LS. The problem of an efficient CA and LS, suiting a particular traffic demand is NP complete [1]. So, we are only concerned with the efficient CA and LS and not in their actual propagation from the central agent to the mesh routers. This can be easily achieved as demonstrated in [5]. The complexity of the CA and LS algorithm is of greater importance in the case of dynamically changing traffic. We propose LoCALS, a polynomial time algorithm, to perform CA and LS. To compare the performance of LoCALS with the optimal solution, we formulate an MILP to get the optimal solution. The main contributions of our work compared to the existing CA and LS approaches are as follows:

- It provides with a polynomially bounded single-pass heuristic, that combines the need for fairness and tries to maximize the total achieved throughput of the network.
- It does not require any specific system support like fast channel switching or any other specific MAC layer support. It can be used with the existing commodity 802.11 based networks without any modifications.
- It decouples CA from routing, thus making the CA and LS algorithms more efficient in terms of computational complexity.
- We show the benefit of considering the traffic demand in CA and LS for maximizing the total achieved throughput by comparing the proposed algorithm with a distributed algorithm that does not consider the traffic demand.

The rest of the paper is organized in the following manner. We discuss our network model and assumptions in Section 2 and then the formal problem statement in Section 3. The MILP formulation is described in Section 4. In Section 5, we propose LoCALS, an efficient heuristic algorithm, for CA and LS. We present the simulation results in Section 6 and conclude the paper in Section 7.

2 Network Model and Assumptions

In this paper, we limit our study only to the mesh routers which form the backbone. Hereafter, whenever we use the term node, we mean a mesh router. The network is given in the form of an *undirected graph* $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents the set of nodes in the network and $\mathcal{E}$ represents the set of edges. G represents the topology of the network, having all *potential* transmissions represented by edges. $\mathcal{V} = \{n_1, n_2, \ldots, n_V\}$ represents the set of nodes in the network. Each node n_i has K_{n_i} radios installed in it. These radios can operate simultaneously independent of each other. Logically, to operate them simultaneously, they need to be tuned to *orthogonal* channels. We assume that $\mathcal{C} = \{c_1, c_2, \ldots, c_C\}$ is the set of orthogonal channels available for the WMN. We denote by V, C, and E the cardinality of the sets $\mathcal{V}$, $\mathcal{C}$, and $\mathcal{E}$ respectively. We assume that the number of channels available in the network is greater than the number of radios in the nodes. The number of available orthogonal channels depends on the radio frequency spectrum used. IEEE 802.11a and IEEE 802.11g support 12 and 3 orthogonal channels, respectively.

An edge $e \in \mathcal{E}$ exists between a node pair $i - j$ *iff* the nodes i and j are in the transmission range of each other. The assumption of an undirected graph forces the transmission ranges of the radio interfaces to be equal. Also, we assume that as long as the nodes i and j are in the transmission range of each other, there is no degradation of the capacity in the transmission between $i - j$ pair with the distance between them. We assume a *two-hop interference model* which serves as a reasonable abstraction of the real scenario [6]. In this model, a transmission in an edge $e = (i, j)$ will interfere with any transmission in *one-hop* and *two-hop* neighbors of both the incident nodes i and j. We denote by F_G the *conflict graph* of G. By conflict graph, we represent the graph with all the edges $e \in \mathcal{E}$

represented by nodes. An edge exists between the nodes $e, e' \in \mathcal{E}$ in the conflict graph if the edges in G interfere with each other.

We assume that a link level synchronization exists among the nodes. The transmission between two nodes takes place only during the time slots assigned to the links. This aids in making the WMN *interference free*, thus avoiding interferences and ensuring proper resource sharing. This property enhances the capacity of the network and gives the provision for unequal sharing of the resources among the interfering edges. Due to the static nature of the WMNs and the presence of a centralized agent, link level synchronization can be achieved. We assume that source-destination pair[1] can transmit data along more than one alternate path simultaneously and that the traffic is *infinitely divisible.* These alternate paths can be found by any existing routing protocols. We assume the existence of a *traffic profiler* which observes the network and makes the centralized agent aware of the traffic demand that exists between every node pair.

3 Problem Statement

We consider the CA and LS problem for a WMN consisting of V nodes and E edges. Each node $n \in \mathcal{V}$ has K_n radios that can function simultaneously and there are C orthogonal channels available. Two nodes i and j can communicate with each other only if $\exists e \in \mathcal{E}$ such that $e = (i, j)$ and both the nodes have at least one radio tuned to a common channel. For the sake of network connectivity, we impose that any CA algorithm must assign a channel to every edge in G, i.e., any two neighbors i and j in G must have at least one radio tuned to a common channel. Note that this is a *sufficient* condition for network connectivity. We denote the CA in the edges by $W_{e,c} \in \{0, 1\}$, where $W_{e,c} = 1$ if the edge $e \in E$ is allocated the channel $c \in \mathcal{C}$. We denote by $Y_{n,c} \in \{0, 1\}$ the radio allocation at the node n. Then $Y_{n,c} = 1$ implies that the node n has a radio tuned to the channel c.

There are a total of T slots for transmission and the links must be scheduled to transmit in these slots. We denote the slot schedules by $X^t_{e,c} \in \{0, 1\}$. $X^t_{e,c} = 1$ if an edge $e \in E$ is assigned a slot $t \in \{1, 2, \ldots, T\}$ in the channel $c \in C$. Note that a link can be scheduled slots only in the channel assigned to it, i.e., $X^t_{e,c} \leq W_{e,c}$ $\forall e \in E$, $\forall c \in C$. The interference graph of the network is given by a *Link Interference Matrix* (*LIM*) [7]. If two edges $a, b \in \mathcal{E}$ interfere with each other then $LIM_{a,b} = 1$ otherwise $LIM_{a,b} = 0$. Due to the assumption on homogeneous transmission ranges, the resulting LIM is a symmetric matrix.

The alternate paths between every source-destination pair, discovered by the routing algorithm, are given by $f^e_{i-j,k} \in \{0, 1\}$ where $i, j \in \mathcal{V}$ and $e \in \mathcal{E}$ and k denotes the k^{th} path between node pair $i - j$. If the k^{th} path between node pair $i - j$ passes through the edge e then the variable $f^e_{i-j,k} = 1$ else $f^e_{i-j,k} = 0$ and any *achieved throughput* between $i - j$ (denoted by x_{i-j}) is given by

[1] Note that we use source-destination pair and node pair interchangeably. In our network, all the nodes are sources and destinations. There are a total of $\binom{V}{2}$ source-destination pairs in the network.

$\sum_k x_{i-j,k}$. We denote the *achieved throughput* in the k^{th} path between node pair $i-j$ by $x_{i-j,k} \geq 0$. The demanded flow between the source-destination pair is given by t_{i-j}. The allocation must also *guarantee* that the minimum throughput achieved must be greater than a minimum fraction (λ) of the demanded flow. This condition ensures the fairness in the network by not neglecting the flows with small demands. The objective of the CA and LS is to maximize the total achieved throughput of the network.

4 MILP Formulation

We adopt a *link model* wherein we allocate channels to edges, keeping in mind the constraint on the number of radios in each node, i.e., the total number of channels assigned to all the edges incident to a node ($n \in \mathcal{V}$) cannot exceed the number of radios in the node (K_n). A link exists between any two nodes which are in the transmission range of each other. According to our assumption, every link must be assigned a channel. This ensures the connectivity of the given

$$\textbf{Maximize: } \sum\sum_{1 \leq i < j \leq V} x_{i-j} \tag{1}$$

Subject to:

$$x_{i-j} \leq t_{i-j} \qquad \forall i, j \in \mathcal{V} \tag{2}$$

$$x_{i-j} \geq \lambda \times t_{i-j} \qquad \forall i, j \in \mathcal{V} \tag{3}$$

$$x_{i-j} \leq \sum_k x_{i-j,k} \qquad \forall i, j \in \mathcal{V} \tag{4}$$

$$\sum_{c \in C} W_{e,c} = 1 \qquad \forall e \in \mathcal{E} \tag{5}$$

$$Y_{n,c} \geq W_{e,c} \qquad \forall e \in \mathcal{E},\ \forall n \in Inc(e) \tag{6}$$

where $Inc(e)$ is the incident nodes of edge e

$$\sum_{c \in C} Y_{n,c} \leq K_n \qquad \forall n \in \mathcal{V} \tag{7}$$

$$X^t_{e,c} \leq W_{e,c} \quad \forall e \in \mathcal{E}, \forall c \in \mathcal{C}, \forall t \in \{1, 2, \ldots, T\} \tag{8}$$

$$X^t_{e,c} + X^t_{e',c} \leq 1 \quad \forall c \in \mathcal{C}, \forall t \in \{1, 2, \ldots, T\}, \forall e, e' \in \mathcal{E},\ LIM_{e,e'} = 1 \tag{9}$$

$$\sum\sum\sum_{1 \leq i < j \leq V,\ k} f^e_{i-j,k} \times x_{i-j,k} \ \leq\ C_{max} \times \frac{\sum\sum_{c \in \mathcal{C}, t \in \{1,2,\ldots,T\}} X^t_{e,c}}{T} \qquad \forall e \in \mathcal{E} \tag{10}$$

Fig. 1. MILP-CALS. MILP formulation for the CA and LS for WMNs with multiple radios and multiple channels.

undirected graph $G = (\mathcal{V}, \mathcal{E})$. The links must also be scheduled to be functional[2] in appropriate slots so that there is no interference in the network. Once the links are given slots, we essentially have shared the resource (channel's bandwidth) in an efficient way. The optimal routes selection (from the available alternate routes) for each node pair needs to be done with the objective of maximizing the total achieved throughput of the network. We denote the MILP formulation for CA and LS as MILP-CALS and is given in Fig. 1.

Eq. 1 represents the objective of maximizing the total achieved throughput. Eq. 2 and Eq. 3 impose the constraints on the achieved throughput. Any value of the achieved throughput above the demand is useless and the achieved throughput must be above a certain minimum fraction (λ) of the demand. The total end-to-end throughput between a node pair $i - j$ is the sum of throughput on each of the alternate paths (Eq. 4). The condition of each edge being assigned exactly one channel is imposed in Eq. 5. The next two equations ensure that the assignment of the channels to the edges is feasible. For an edge to be assigned a channel, there must be at least one radio on both the incident nodes that are tuned to this channel (Eq. 6). Also, number of distinct channels that can be assigned to the radios on a node cannot exceed the total number of radios at that node (Eq. 7). Eq. 8 and Eq. 9 ensure that no two interfering edges, that are assigned the same channel, be assigned the same slot. Eq. 10 represents the capacity constraint. The RHS denotes the total capacity supported on an edge and the LHS denotes the sum of all flows passing through the edge.

5 Channel Assignment and Link Scheduling Algorithm

The MILP can be solved for relatively small networks. But as the size of the network increases the computational complexity increases exponentially. More often the problem becomes intractable. This renders the MILP formulation practically incapable of handling real life networks. So, we propose a polynomially bounded algorithm for load aware CA and LS, **LoCALS**, which performs comparable to the optimal solution.

The algorithm addresses the problem of CA, LS, and flow allocation separately. The key idea behind LoCALS is that the CA and LS is done using the expected loads on the edges. We calculate the expected loads on the edges based on the traffic demand. On the basis of these expected loads, CA is done in order to provide diverse set of channels to the "hot-spots" where the expected load on the edges are high. The bandwidth allocation to the edges is done on the basis of the relative proportions of the expected loads during LS. Finally, we solve an LP (polynomially bound) [8]) for final route allocation and to compute the total achieved throughput. The routing algorithm gives a set of alternate paths that are represented by the binary variables $f^{e}_{i-j,k}$. Now, each node pair can choose a subset of the available alternate paths to send the traffic. The optimal subset selection can be done only after the CA and LS. But the CA algorithm is

[2] By functional, we mean that the links can be used to send and receive in their allotted slot. The direction of the traffic for that slot is decided by the incident nodes.

highly dependent on the initial set of routes selected. This leads to a cyclic inter-dependency between CA and route selection. We break this inter-dependency by initially including all the possible alternate routes for each node pair for the purpose of CA. We assign probabilities to the routes using the hop-count of each path, where the probability denotes the likelihood of the particular route to be included in the final route selection.

As the hop-count of the path increases, the probability of choosing the path for the final solution must decrease and the path with least hop-count must be given higher probability. Let $Path_{i-j,k} = \{e| \ f^e_{i-j,k} = 1\}$ denote the set of edges $e \in E$ that belong to the k^{th} path between the node pairs $i - j \in \mathcal{V}$. Let us define a quantity $\alpha_{i-j,k}$ for each path between node pairs $i - j$, such that

$$\alpha_{i-j,k} \quad = \quad \begin{cases} \frac{1}{|Path_{i-j,k}|} \ if \ |Path_{i-j,k}| \neq 0 \\ 0 \qquad\qquad otherwise \end{cases} \tag{11}$$

The *expected load* on each path is given by $\mathcal{P}_{i-j,k} \times t_{i-j}$, where $\mathcal{P}_{i-j,k}$ is the probability assigned to each path $(\frac{\alpha_{i-j,k}}{\sum\limits_{k'} \alpha_{i-j,k'}})$. With the expected load on each path for each node pair $i - j \in \mathcal{V}$, the expected load on each edge is given by

$$Exp(e) \quad = \quad \sum_{1 \leq i < j \leq V} \sum_{k} f^e_{i-j,k} \times \mathcal{P}_{i-j,k} \times t_{i-j} \tag{12}$$

These loads give us means of assigning weights to the edges proportional to load. The CA and LS use these weights to assign channels and slots, respectively. We also define another useful quantity $Exp_{Reg}(e)$ for each edge e. This captures the expected load on all the edges $e' \in E$ that could potentially interfere with the edge e.

$$Exp_{Reg}(e) \quad = \quad Exp(e) + \sum_{e' \in \mathcal{E}} LIM_{e,e'} \times Exp(e') \tag{13}$$

5.1 Channel Assignment

The underlying objective of our CA algorithm is to minimize the co-channel interference among edges with high expected load. Note that the CA depends on the demanded flow, because the expected load of the edges changes with demanded flow. The CA algorithm is described in Algorithm 1.

We construct a new set $\mathcal{E}'$ which is initialized to $\mathcal{E}$ and an empty set $SatNodes$. As the edges are assigned channels, they are removed from the set $\mathcal{E}'$. This is repeated until all the edges are assigned a channel. From lines 6-20, we find a new edge $e \in \mathcal{E}'$ with the highest $Exp_{Reg}(e)$. The edge with the highest $Exp_{Reg}(e)$ represents the edge with highest potential to interfere with other edges and decrease the throughput. In case there is a tie in this value, we choose the edge with the highest expected load. Once the channel is assigned to an edge, it is likely that the last radio in one (both) of the incident node(s) is utilized for

Algorithm 1. Channel Allocation

```
1.  **Input:** A Graph $G = (\mathcal{V}, \mathcal{E})$ of $E$ links, and $Exp(e)$ for each link $e \in \mathcal{E}$
2.  **Output:** Channel Allocation for edges, $W_{e,c}$
3.  Set $W_{e,c} \leftarrow 0 \quad \forall e \in \mathcal{E},\ c \in \mathcal{C}$    /*Initializing the $W_{e,c}$ variables */
4.  Let $\mathcal{E}' \leftarrow \mathcal{E}$, $c_x \leftarrow c_1$, $SatNodes \leftarrow \Phi$
5.  **while** $\mathcal{E}'$ is not empty **do**
6.    **if** $SatNodes$ is empty **then**
7.      /* Finding the edge which is maximally interfered */
8.      Find the edge $e \in \mathcal{E}$ in $G'$ with largest $Exp_{Reg}(e)$. The edge with higher load is chosen in
        case of a tie.
9.      /* Calculating the current load on each channel in edge e's neighborhood */
10.     $Load(e,c) = \sum_{e'} LIM_{e,e'} \times W_{e',c} \times Exp(e') \quad \forall c \in \mathcal{C}$
11.     Choose $c$ with the least $Load(e,c)$, with preference to lower index of $c$ to break tie
12.     $c_x \leftarrow c$, $W_{e,c_x} \leftarrow 1$, $\mathcal{E}' \leftarrow \mathcal{E}' - \{e\}$    /*Assigning channel $c_x$ to edge $e$ */
13.     $Y_{i,c_x} \leftarrow 1,\ \ Y_{j,c_x} \leftarrow 1$ where $e = (i,j)$    /*Assigning the chosen channel to the
        incident nodes */
14.     /*Checking for saturation of radios on the incident nodes*/
15.     **if** $\sum_{c'=1}^{C} Y_{i,c'} = K_i$ **then**
16.       $SatNodes \longleftarrow SatNodes \cup \{i\}$
17.     **end if**
18.     **if** $\sum_{c'=1}^{C} Y_{j,c'} = K_j$ **then**
19.       $SatNodes \longleftarrow SatNodes \cup \{j\}$
20.     **end if**
21.   **else** $\{SatNodes$ is not empty$\}$
22.     /*Assigning the remaining edges of the currently saturated nodes with the last channel
        chosen */
23.     **while** $SatNodes$ is not empty **do**
24.       Set $i \leftarrow GetOneElement(SatNodes)$
25.       Construct $UnassignedEdges(i) = \{e|e \in \mathcal{E}, i \in Inc(e),\ W_{e,c} = 0\ \forall c \in \mathcal{C}\}$
26.       **for all** $e \in UnassignedEdges(i)$ **do**
27.         /*Assigning the channel $c_x$, the last assigned channel to the other edges*/
28.         $W_{e,c_x} \leftarrow 1$, $SatNodes \leftarrow SatNodes - \{e\}$, $Y_{i,c_x} \leftarrow 1$, $Y_{j,c_x} \leftarrow 1$ where $e = (i,j)$
29.         /*Checking if the other incident node is saturated*/
30.         **if** $\sum_{c'=1}^{\mathcal{C}} Y_{j,c'} = K_j$ **then**
31.           $SatNodes \longleftarrow SatNodes \cup \{j\}$
32.         **end if**
33.       **end for**
34.       $SatNodes \leftarrow SatNodes - \{i\}$
35.     **end while**
36.   **end if**
37. **end while**
```

this assignment, in which case the node(s) is(are) called *saturated.* We add a saturated node to the set $SatNodes$. When the loop at line 5 is entered again, if the $SatNodes$ is non empty, then there are some nodes that became *saturated* in the previous step. This implies that the links that are incident on these saturated nodes and are yet to be assigned any channel, cannot be assigned a new channel. These unassigned links are then assigned *that* channel, whose assignment to one of the neighboring links in the previous step, led to the saturation of the node (lines 24-34).

5.2 Link Scheduling

Once the channels are allocated, the LS must be done for proper resource sharing. We provide a greedy heuristic algorithm for LS. The slots must be allotted in such a way that any two interfering links sharing a common channel must not be given the same slot. Hence, the total of T slots must be divided among the

Algorithm 2. Link Scheduling

```
Input: $LIM$, $W$, $Frac(e)$, $T$ and conflict graph $F_G$
Output: The slot schedules, $X^t_{e,c}$
Sort the edges in the decreasing order of their degree. Let $(e_1, e_2, \ldots, e_E)$ denote the sorted order
for $i = 1$ to $E$ do
  $N(e_i) = ceil(T \cdot Frac(e_i))$, the maximum number of time slots $e_i$ will be active
  Let $e_i = (u, v)$. $allocated \leftarrow 0,\ t \leftarrow 0$.
  Let $c \in \mathcal{C}$ be the channel allocated to edge $e_i$
  while $allocated \leq N(e_i)$ and $t \leq T$ do
    if $X^t_{e',c} = 0\ \ \forall e' \in \mathcal{E}$ and $LIM_{e,e'} = 1$ then
      $X^t_{e,c} \leftarrow 1,\ allocated++,\ t++$
    end if
  end while
end for
```

$$Maximize : \sum\sum_{1 \leq i < j \leq V} x_{i-j}$$

Subject to:

$$x_{i-j} \leq t_{i-j} \qquad \forall i, j \in \mathcal{V}$$

$$x_{i-j} \geq \lambda \times t_{i-j} \qquad \forall i, j \in \mathcal{V}$$

$$x_{i-j} \leq \sum_{k} x_{i-j,k} \qquad \forall i, j \in \mathcal{V}$$

$$\sum\sum\sum_{1 \leq i < j \leq V,\ k} f^e_{i-j,k} \times x_{i-j,k} \ \leq\ C_{max} \times \frac{\sum\sum_{c \in \mathcal{C}, t \in \{1,2,\ldots,T\}} X^t_{e,c}}{T} \qquad \forall e \in \mathcal{E}$$

Fig. 2. LoCALS - Flow allocation (LP)

interfering links sharing the same channel. The division can be made on the basis of the expected loads calculated for each edge. We calculate the tentative fraction of channel that needs to be provided to each edge in the following manner:

$$Frac(e) \quad = \quad \frac{Exp(e)}{\sum_{e' \in \mathcal{E}} LIM_{e,e'} \times W_{e',c} \times Exp(e')}, \tag{14}$$

where $e \in \mathcal{E},\ c \in \mathcal{C},$ and $W_{e,c} = 1$.

Essentially the number of slots a link can potentially get is proportional to its expected load. The number of slots can be calculated by $N(e) = \lceil T \times Frac(e) \rceil$. Since we are using a ceiling function, this indicates the maximum number of slots the link e can be given. Our LS algorithm is presented in Algorithm 2. The edges are sorted in the descending order of their interference degree in their conflict graph F_G. We start allocating slots from the edge with the highest degree. Note that this is a greedy algorithm and sometimes it could lead to poor LS. For each edge e, we check its interfering edges for free slots and assign them to the current edge e.

5.3 Flow Allocation

After the first two phases of the heuristic algorithm, we get the binary variables $W_{e,c}$ and $X^t_{e,c}$ for the original MILP formulation given in Fig. 1. They are found using probabilistic methods by assigning expectation to the selection of a particular route (Eq. 11, 12). Now the actual route selection (from the set of alternate routes provided by the routing algorithm) and flow allocation (decisions on how much traffic to send through each selected route) need to be done. This problem is similar to the MILP formulation except that the binary variables are already decided. Hence, we reformulate the problem with the relevant constraints in Fig. 2 that has only linear variables. The amortized complexity of an LP is bound by polynomial time [8].

6 Simulation Results

In this section, we study the performance of LoCALS. We compare the optimal solution obtained through MILP-CALS with the solution obtained through LoCALS on a 2 × 3 grid network. We then compare LoCALS with a load unaware CA algorithm called SAFE [9] on a 7 × 7 grid topology. Finally, we study the effect of the number of radios and channels on the performance of LoCALS. In what follows, we define the *load* as the sum of the demands between every node-pair ($\sum\sum_{1 \le i < j \le V} t_{i-j}$). The value of λ in the minimum guarantee constraint (Eq. 3) is fixed at 0.01 in all the simulations. In each simulation, 10 random traffic patterns are generated and the results are averaged over these 10 traffic matrices.

6.1 MILP-CALS vs LoCALS

Here, we compare the performance of LoCALS with the optimal solution on a 2 × 3 grid topology with 2 radios, 3 channels, and maximum channel capacity of 10 Mbps for each channel. From Fig. 3, we find that LoCALS performs on par with the optimal solution. As the load increases, the achieved throughput increases which is quite intuitive. When the load is low (below 15 Mbps), the network is able to serve the highest traffic demand, but as the traffic demand increases further, the achieved throughput is unable to match the highest traffic demand. This trend is noticed in with LoCALS starting from load of 15 Mbps. The MILP is becoming intractable for more reasonable scenarios with bigger networks and hence, extensive comparisons could not be done.

6.2 Comparison with Load Unaware Algorithm (SAFE)

In order to demonstrate the need for load awareness in the context of efficient utilization of network resources, we perform comparisons of LoCALS with SAFE, a load unaware CA algorithm. One important thing to note is, LoCALS does both CA and LS, whereas SAFE only does the CA. For the sake of comparison,

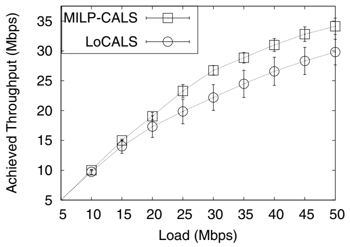


Fig. 3. LoCALS vs MILP-CALS. 2×3 grid with 2 radios and 3 channels.

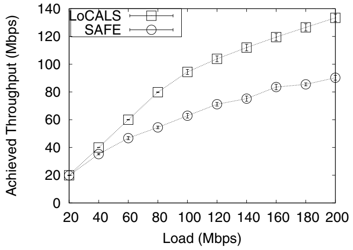


Fig. 4. LoCALS vs SAFE. 7 × 7 grid with 3 radios and 5 channels.

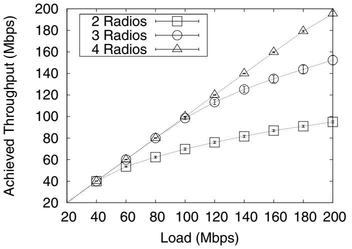


Fig. 5. Variation of radios with 6 channels

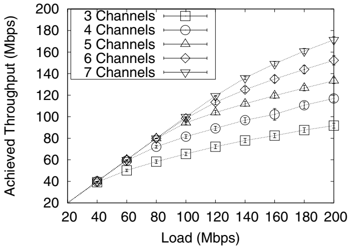


Fig. 6. Variation of channels with 3 radios

we assign slots to the links after the CA is done using SAFE. The general assumption made in most of the works that does not explicitly dealing with LS or bandwidth allocation, is that the channel bandwidth is split equally among the links that share the same channel. Hence, we modify our LS to accommodate this criterion in assigning the slots for SAFE's CA algorithm. In our LS, we set the expected load of each link to be of unit value. Under this setting, the LS algorithm essentially tries to split the channel bandwidth equally among the interfering links sharing the same channel. As SAFE is a randomized algorithm, the performance depends upon the random seed to a great extent. Hence, for each scenario, we average the results of SAFE over 25 seeds. We take a 7 × 7 grid network and the routing algorithm gives 5 paths of the least hop length between any two node pairs. Fig. 4 shows the achieved throughput for 3 radios and 5 channels. It can be clearly seen that LoCALS out-performs SAFE as the load in the network increases.

6.3 Impact of Number of Radios and Channels

With increasing radios and channels, we expect the overall network capacity to increase and thus higher achieved throughput. We conduct the simulations on the 7×7 grid. Fig. 5 and Fig. 6 show the impact of increasing radios and channels, respectively, on LoCALS. In Fig. 5, we fix the number of orthogonal channels in the network to be 6 and vary the number of radios, whereas in Fig. 6, we fix the number of radios to 3 and vary the number of channels. For lower loads, there is not much of an impact for increase in radios as well as channels. However, as the load increases, the increased network capacity is utilized by LoCALS to provide better channel allocations with higher achieved throughput.

7 Conclusion

WMNs with the potential to integrate with several kinds of external network is the future of last mile networks. The provision of the routers possessing multiple radios and the existence of multiple orthogonal channels has led to improved capacity of WMNs thus making it more relevant to replace the wired counterparts in the LANs. We have proposed an MILP formulation and a polynomially bound heuristic algorithm for efficient channel assignment and link scheduling given the prior traffic demands. Our results indicate that the heuristic algorithm is quite comparable to the optimal solution. Our work combines the need for fairness and maximizing the total achieved throughput of the network. Our work does not require any special modifications to the existing MAC protocols. In the future, we would like to investigate the effect of changing traffic demands, in terms of the cost of reconfigurations and efficient utilization of underlying network resources.

References

1. Raniwala, A., Gopalan, K., Chiueh, T.: Centralized Channel Assignment and Routing Algorithms for Multi-Channel Wireless Mesh Networks. ACM SIGMOBILE Mobile Computing and Communications Review 8(2), 50–65 (2004)
2. Bahl, P., Chandra, R., Dunagan, J.: SSCH: Slotted Seeded Channel Hopping for Capacity Improvement in IEEE 802.11 Ad Hoc Wireless Networks. In: Proceedings of the 10th Annual International Conference on Mobile Computing and Networking (MobiCom 2004), October 2004, pp. 216–230 (2004)
3. Tang, J., Xue, G., Zhang, W.: End-to-End Rate Allocation in Multi-Radio Wireless Mesh Networks: Cross-Layer Schemes. In: Proceedings of the ACM International Conference on Quality of Service in Heterogeneous Wired/Wireless Networks (QShine, 2006) (2006)
4. Alicherry, M., Bhatia, R., Li, L.E.: Joint Channel Assignment and Routing for Throughput Optimization in Multi-Radio Wireless Mesh Networks. In: Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom 2005), August 2005, pp. 58–72 (2005)
5. Ramachandran, K.N., Belding, E.M., Almeroth, K.C., Buddhikot, M.M.: Interference-Aware Channel Assignment in Multi-Radio Wireless Mesh Networks. In: Proceedings of the IEEE International Conference on Computer Communications (INFOCOM 2006), April 2006, pp. 1–12 (2006)
6. Kumar, V.S.A., Marathe, M.V., Parthasarathy, S., Srinivasan, A.: End-to-End Packet-Scheduling in Wireless Ad Hoc Networks. In: Proceedings of the ACM-SIAM symposium on Discrete algorithms (SODA 2004), January 2004, pp. 1021–1030 (2004)

7. Das, A.K., Alazemi, H.M.K., Vijayakumar, R., Roy, S.: Optimization Models for Fixed Channel Assignment in Wireless Mesh Networks with Multiple Radios. In: Proceedings of the IEEE Communication Society Conference on Sensor and Ad Hoc Communications and Networks (SECON 2005), September 2005, pp. 463–474 (2005)
8. Gonzaga, C.C.: On the Complexity of Linear Programming. Resenhas - IME-USP Journal, special issue dedicated to Paul Erdos 2(2), 197–207 (1995)
9. Shin, M., Lee, S., Kim, Y.: Distributed Channel Assignment for Multi-Radio Wireless Networks. In: Proceedings of the IEEE International Conference on Mobile Adhoc and Sensor Systems (MASS 2006), October 2006, pp. 417–426 (2006)

Multi-round Real-Time Divisible Load Scheduling for Clusters*

Xuan Lin, Jitender Deogun, Ying Lu, and Steve Goddard

Department of Computer Science and Engineering
University of Nebraska - Lincoln, Lincoln, NE 68588
{lxuan,deogun,ylu,goddard}@cse.unl.edu

Abstract. Quality of Service (QoS) provisioning for divisible loads in cluster computing has attracted more attention recently. To enhance QoS and provide performance guarantees in cluster computing environments for divisible loads, in this paper, we integrate a Simplified Multi-Round (SMR) strategy into the design of real-time scheduling algorithms for divisible load applications. Four contributions are made in this paper. First, we present algorithm SMR and extend it to compute a closed form formula for minimum number of processors required to meet an application deadline. Second, we derived a closed form solution for execution time of the optimized SMR. Third, we formally prove that optimized SMR results in better completion time than the single round strategy. Finally, we integrate SMR with our algorithm framework and propose two sets of efficient algorithms.

1 Introduction

Scheduling parallel applications in distributed computing resources has been studied extensively for a variety of application models, such as the well-known directed acyclic task graph model. Another type of model is arbitrary divisible load application model. Arbitrarily divisible applications consist of an amount of data that can be divided arbitrarily into any number of independent load fractions, and each fraction itself is arbitrarily divisible. The arbitrarily divisible applications represent problems of great significance for cluster-based research computing facilities such as the U.S. CMS (Compact Muon Solenoid) Tier-2 sites [1] , which are associated with the Large Hadron Collider (LHC) at CERN (European Laboratory for Particle Physics).

Divisible Load Theory (DLT) initially motivated by the objective of integrating communication and computation in distributed sensor networks, was developed to determine load distribution strategies for arbitrarily divisible loads in distributed computing environments [2,3]. A great deal of significant progress has been made in the *Divisible Load Theory* (DLT). However, all of the work focuses on scheduling a single divisible-load task. And the goal, is usually to minimize

* This work was partially supported by USDA FCIC/RMA 2IE08310228 and NSF CNS 0720810.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 196–207, 2008.

the *makespan*, that is, to minimize the execution time of a task. Morever, the amount of computing resources to be assigned to a task is usually assumed to be known before scheduling.

In recent years, efficient utilization of large-scale computing resources like clusters, for providing QoS guarantees, has become more and more important. In our recent work [4,6,5], we are the first to propose cluster-based, real-time scheduling algorithms which integrate *divisible load theory* [3] and real-time scheduling algorithms for online scheduling of divisible load applications to provide QoS. Evaluations show that integrating DLT into real-time scheduling algorithms outperforms other FIFO and EDF based algorithms. However, in our previous work we only address a single-round scheduling strategy, where the load is divided and distributed to the processing nodes in a single round. In single round scheduling, if the application is data intensive, the processing nodes may face long idle times while waiting for data transmission. Multi-round strategies [2,7], have been proposed to solve this problem. By subdividing the fractions of data further and distributing them in a repetitive sequence, the multi-round strategy incorporates pipelining and reduces the processor idle time. However, as discussed above, these work only consider a single task. While for online scheduling of multiple tasks, the problem becomes much more complicated. In this paper, we investigate how to integrate multi-round strategy into real-time scheduling of divisible loads to provide QoS guarantees. Four contributions are made in this paper. First, a multi-round algorithm UMR (Uniform Multi-Round) [7] is modified and extended to develop SMR (Simplified Multi-Round) algorithm. SMR is designed to fit into our model and is extended to compute the minimum number of processors required to meet an application's deadline, which is a critical factor for the real-time scheduling algorithm to determine the task's partition and schedule. Second, we formally prove that optimized SMR results in better completion time than the single round strategy. Third, we derive a closed form formula for the execution time of the optimized SMR. Finally, we integrate SMR with our algorithm framework and propose and evaluate two sets of efficient algorithms.

The remainder of this paper is organized as follows. Related work is presented in Section 2. We describe both task and system models in Section 3. In Section 4 we discuss real-time scheduling algorithms investigated in this paper. We evaluate the performance of algorithms in Section 5 and conclude the paper in Section 6.

2 Related Work

Development of clusters and Grid computing have recently gained considerable momentum. By linking a large number of computers together, a cluster provides cost-effective power for solving complex problems. In a large-scale Grid, a resource management system (RMS) is central to its operation. In order to serve end-users in a timely fashion, it is essential for the underlying cluster RMS to provide performance guarantees or QoS.

Research has been carried out in utility-driven cluster computing [8] to improve the value of utility delivered to the users. Proposed cluster RMSs [9] have addressed the scheduling of both sequential and parallel workloads. The goal of these schemes is similar to ours—to harness the power of resources based on user objectives.

Divisible load theory [2,3,7,10] provides an in-depth study of distribution strategies for arbitrarily divisible loads in multiprocessor/multicomputer systems subject to system constraints like link speed, processor speed and interconnection topology. The goal of divisible load theory is to exploit parallelism in computational data so that the workload can be partitioned and assigned to several processors such that execution completes in the shortest possible time [2]. The DLT has extensive application [10]. An example related to our work is its applications [11,12] and implementation in Grid computing [13]. However, all the previous work on DLT focuses on minimizing the makespan of a single application, while we deal with multiple, dynamic tasks in the system. Besides, our goal is to provide QoS for the whole system instead of optimizing one single application.

In our previous work [5,4,6], we applied single round divisible load strategy to the design of real-time scheduling algorithms for cluster computing; specifically, divisible load theory is applied to the scheduling of applications, such as CMS [1], to provide QoS. As discussed in [2], for a single task, multi-round algorithm will have better performance than the single round algorithm. However, in our scenario, we have multiple, dynamic tasks arriving to the system. It is still unknown whether integrating multi-round strategy has any advantages in the context of providing QoS guarantees. Moreover, the multi-round algorithm discussed in [2] are complicated and can not be easily implemented. Thus, in [14], Y. Yang et. al. proposed an algorithm UMR. In this paper, we extended our previous work by integrating SMR, a simplified and extended version of UMR, into our algorithmic framework.

3 Task and System Models

In this section, we present our task and system model.

Task Model. We investigate real-time scheduling of arbitrarily divisible tasks that arrive aperiodically and execute non-preemptively (once subtasks are allocated to processors). In our model, a divisible task $T_i = (A_i, \sigma_i, D_i)$ is a single invocation, where A_i is the arrival time of the task, σ_i is the total data size of the task, and D_i is the relative deadline. Task execution time is dynamically determined using σ_i and allocated resources—processing nodes and bandwidth—by leveraging the modeling power of divisible load theory [3], as explained in Section 4.

System Model. A cluster consists of head node, denoted P_0, and N processing nodes, denoted by $P_1, \ldots, P_N$. The system model assumes a typical cluster environment in which the head node does not participate in computation. The role of the head node is to accept or reject incoming tasks, execute the scheduling algorithm, and divide and distribute the workload to processing nodes. A star network topology is used to represent the communication requirements of the cluster. Since tasks and subtasks are independent, there is no need for processing nodes to communicate with each other.

In this work, a homogenous cluster and sequential transmission of the workload is assumed. Thus, (1) all processing nodes have the same computational power; (2) all links from the head node to the processing nodes have the same bandwidth; and (3) in each round, the head node begins to distribute the workload to node P_{i+1} only after it has completed its workload transmission to node P_i.

As with DLT, we use linear models to represent processing and transmission times [3]. In the simplest scenario, the computation time of a load σ is calculated by a cost function $\sigma\chi$, where χ represents the time to compute a unit of workload on a single processing node. The communication time of a load σ is calculated by a cost function $\sigma\tau$, where τ is the time to transmit a unit of workload from the head node to a processing node. For many applications the output data is just a short message and is negligible, particularly considering the very large size of the input data. Therefore, in this paper we only model the transfer of application input data but not the transfer of output data. The extension to consider the output data transfer using DLT is straightforward.

The following notations, partially adopted from [3], are used in this paper,

- $T_i = (A_i, \sigma_i, D_i)$, where A_i is the arrival time of the task, σ_i is the total data size of the task, and D_i is the relative deadline;
- $\alpha = (\alpha_1, \alpha_2, ..., \alpha_n)$: Data distribution vector, where n is the number of processing nodes allocated to the task, α_j is the data fraction allocated to the j^{th} node, i.e., $\alpha_j\sigma$, is the amount of data that is to be transmitted to the j^{th} node for processing, $0 < \alpha_j \leq 1$ and $\Sigma_{j=1}^{n}\alpha_j = 1$;
- τ: Cost of transmitting a unit workload;
- χ: Cost of processing a unit workload.
- n^{min}: Minimum number of processors needed by a task to meet its deadline.

4 Algorithms

In Section 4.1, we first briefly discuss our algorithm framework and then we present SMR, a simplified and extended version of UMR [7] algorithm. We then describe how to integrate SMR into our algorithm in Section 4.2. The minimum number of nodes to be assigned to a task is derived as well as the number of rounds we need to deliver the workloads. Finally, we prove that applying SMR will result in a better execution time for a task in Section 4.3.

4.1 Algorithm Framework

As is typical for dynamic real-time scheduling algorithms [15,16], when a task arrives, the scheduler dynamically determines if it is feasible to schedule the new task without compromising the guarantees for previously admitted tasks. Upon completion of the *schedulability test*, if all tasks are schedulable a feasible schedule is developed and the new task is accepted, otherwise, it is rejected.

The general framework for a *schedulability test* is discussed in our previous work [4]. The framework can be configured to generate various real-time divisible load scheduling algorithms by configuring the following three components of *schedulability test*:

1. Scheduling policy (e.g., FIFO, EDF).
2. Node assignment method (assigning a task all N or its n^{min} nodes).
3. Task partitioning rule (single-round DLT or SMR).

In [4], we have already investigated single-round algorithms and show that algorithms that assign n^{min} nodes have their advantages. Thus, in this paper, for the second and third components, we only investigate algorithms with n^{min} nodes and SMR. First two algorithms, FIFO-MN-SMR and EDF-MN-SMR, are developed by configuring the first module to adopt FIFO and EDF accordingly. By applying the optimized algorithm described in Section 4.2, we have two optimized algorithms, FIFO-MN-SMR* and EDF-MN-SMR*.

4.2 Simplified Multi-Round (SMR)

In this subsection, we describe how to integrate a simplified UMR [7], a multi-round DLT, into our algorithmic framework. In subsection 4.2, we briefly describe the SMR algorithm, a simplified version of UMR. UMR was first introduced in [7] and one of its motivations is to reduce the complexity of the original multi-round DLT algorithm introduced in [2]. In this paper, our system model is different since we do not consider the setup cost. We modify the UMR algorithm to fit into our model and thus develop a simplified multi-round approach, named SMR. In Subsection 4.2, we derive a formula for the minimum number of nodes to be assigned to a task, which is a critical factor for the real-time scheduling algorithms to determine the task partitioning and schedule. In [14], an optimization of UMR is proposed, but no details of the derivation of the task execution time are given. However, we present a detailed derivation for our model (no setup cost is considered) in Subsection 4.2. We also derive the minimum number of nodes and the number of rounds for the optimized algorithm.

Illustration of SMR. SMR is a multi-round scheduling algorithm that is based on simplifying UMR [7]. It dispatches a load in many rounds, and in each round the load fraction is further equally subdivided among all the nodes. Let M denote the number of rounds used by UMR/SMR. Figure 1 shows the operation of SMR in the j^{th} and $j+1^{th}$ round (for $j \in \{1, 2, \cdots M-1\}$). The approach is presented in detail in [7]. And since in our model we do not consider the setup cost, the notion and equations in the figures are modified accordingly. At time T_0, the head node begins to dispatch chunks of size CH_j for the j^{th} round. At T_1 node P_1 receives its data for the j^{th} round and begins execution, and at time T_2 node P_2 receives its data for that round and begins to compute. At time T_3,

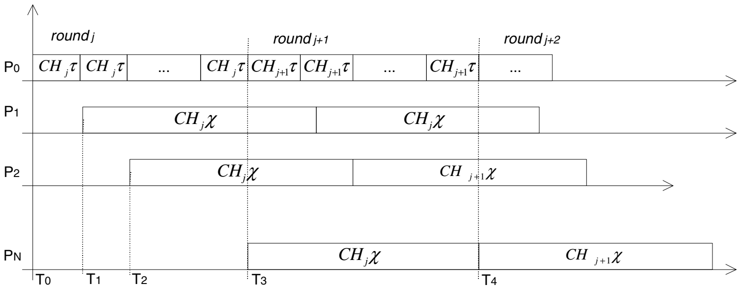


Fig. 1. Timing Diagram of SMR

the head node finishes dispatching data for the j^{th} round and begins to send data for the $j + 1^{th}$ round.

According to [14], utilization of processing nodes is maximized if the time needed by processing node P_N to compute the j^{th} round load is equal to the time used by the head node to transmit the load for the $j+1^{th}$ round (in Figure 1, it is the time interval between T_3 and T_4). That is, $\forall j \in \{1, 2, \cdots M - 1\}$

$$CH_j \times \chi = N \times CH_{j+1} \times \tau. \tag{1}$$

By simple transformation, we get

$$CH_{j+1} = \frac{\chi}{N \times \tau} \times CH_j. \tag{2}$$

This equation gives the relation of chunk sizes for consecutive rounds. It implies that for the SMR strategy once the data chunk size for the first round is determined, the data chunk sizes for all other rounds can be derived following equation (2).

Minimum Number of Nodes Analysis. Unlike DLT, where the number of processing nodes allocated to a task is assumed to be given, we derive the minimum number of nodes needed to meet the task deadline, which is a critical factor for the real-time scheduling algorithm to determine the task partitioning and schedule.

Observe the behavior of the algorithm in Figure 1, we notice that P_N is the last processing node to complete its computation. Therefore, the task's completion time is the time when node P_N completes its computation for the last round. The task's execution time is the difference between its completion and start times. In SMR, data sent to all the nodes has the same size. Thus, the processing time on a node is equal to $\frac{\sigma}{N}\chi$. The total execution time of the task is $\frac{\sigma}{N}\chi + N \times CH_1 \times \tau$, where the first term is the processing time and the second term is the idle time of node P_N waiting for the data of the first round to arrive. That is,

$$\mathcal{E} = \frac{\sigma}{N}\chi + N \times CH_1 \times \tau \tag{3}$$

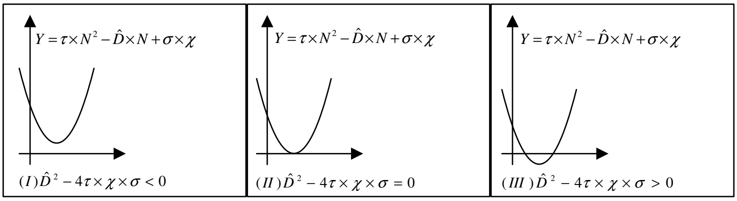


Fig. 2. Parabola $Y = \tau \times N^2 - \hat{D} \times N + \sigma \times \chi$

By analyzing the above equation, we conclude that the minimum execution time of a task scheduled by SMR on N processing nodes is achieved when the chunk size for the first round, CH_1, is infinitely small.

Assuming the minimum chunk size is a unit of workload, equation (3) becomes

$$\mathcal{E} = \frac{\sigma}{N}\chi + N \times \tau \tag{4}$$

If the task $T = (A, \sigma, D)$ has start time s, then the completion time $\mathcal{C} = s + \mathcal{E} \leq A + D$, because the task must satisfy its deadline. It follows that,

$$\mathcal{E} = \frac{\sigma}{N}\chi + N \times \tau \leq A + D - s \tag{5}$$

Let $\hat{D} = A + D - s$. The above inequality is equivalent to

$$\tau \times N^2 - \hat{D} \times N + \sigma \times \chi \leq 0 \tag{6}$$

Since $\tau > 0$, $Y = \tau \times N^2 - \hat{D} \times N + \sigma \times \chi$ is a parabola that opens upward. See Fig. 2.

Fig. 2 shows three positions of parabola Y corresponding to negative, zero and positive value of $\hat{D}^2 - 4\tau\chi\sigma$. To derive the minimum N that will satisfy constraint (6), we need to analyze the three cases.

In the first case, when $\hat{D}^2 - 4\tau\chi\sigma < 0$, the parabola has no real axis intercepts, which implies that the value of $\tau \times N^2 - \hat{D} \times N + \sigma \times \chi$ will always be greater than 0. Therefore constraint (6) can not be satisfied for any real number N, implying it is not possible to meet the task deadline.

In the second case, when $\hat{D}^2 - 4\tau\chi\sigma = 0$, the parabola has only one real axis intercept where $N = \frac{\hat{D}}{2\tau}$. This is the only possible value of N that satisfies constraint (6). In addition, N, the number of processing nodes, must be a positive integer. Thus, the task can meet its deadline if and only if $N = \frac{\hat{D}}{2\tau}$ is a positive integer.

In the third case, when $\hat{D}^2 - 4\tau\chi\sigma > 0$, the parabola has two real axis intercepts. From Figure 2, we can see that in order to satisfy (6), the value of N should fall between the two real roots of equation $\tau \times N^2 - \hat{D} \times N + \sigma \times \chi = 0$. That is

$$\frac{\hat{D}-\sqrt{\hat{D}^2-4\tau\chi\sigma}}{2\tau} \leq N \leq \frac{\hat{D}+\sqrt{\hat{D}^2-4\tau\chi\sigma}}{2\tau}.$$

Since N must be a positive integer, we can obtain a reasonable minimum number for N only if the following constraint holds.

$$1 \leq \left\lceil \frac{\hat{D}-\sqrt{\hat{D}^2-4\tau\chi\sigma}}{2\tau} \right\rceil \leq \frac{\hat{D}+\sqrt{\hat{D}^2-4\tau\chi\sigma}}{2\tau}.$$

Satisfying the above constraint, the minimum number of nodes a task needs to complete before its deadline is

$$N = \left\lceil \frac{\hat{D}-\sqrt{\hat{D}^2-4\tau\chi\sigma}}{2\tau} \right\rceil.$$

Last Round Optimization. V. Bharadwaj et al. [2] show that in an optimal divisible workload partition, all nodes finish computing at the same time. As discussed above, for each round, each node is assigned chunks of the same size. Thus, the computing nodes finish at different times. Morever, when the τ is large, that is, when the communication cost is high, the first node will finish computation much earlier than the last. This problem is fixed by repartitioning the chunks in the last round. Thus, in the optimized SMR, the chunk size for each node is the same in all rounds except the last round.

After the optimization, we have

$$\mathcal{E}^* = \frac{\sigma}{N}\chi + \frac{N}{2}\tau \tag{7}$$

Now, applying the same process as in Sec. 4.2, we can calculate the minimum number of nodes N^* as follows:

$$N^* = \begin{cases} \frac{\hat{D}}{\tau} & \text{if } \hat{D}^2 - 2\sigma\tau\chi = 0 \text{ and } \frac{\hat{D}}{\tau} \text{ is an interger,} \\ \lceil \frac{\hat{D}-\sqrt{\hat{D}^2-2\tau ps\sigma}}{2\tau} \rceil & \text{if } \hat{D}^2 - 2\sigma\tau\chi > 0 \text{ and} \\ & 1 \leq \lceil \frac{\hat{D}-\sqrt{\hat{D}^2-2\tau ps\sigma}}{2\tau} \rceil \leq \frac{\hat{D}+\sqrt{\hat{D}^2-2\tau ps\sigma}}{2\tau}. \end{cases} \tag{8}$$

4.3 The Optimized SMR Theorem

Intuitively, the multi-round algorithm will perform better than the single-round algorithm since the data transmission and computation are better pipelined. However, for UMR [7], no formal proof is given. Thus, in this section, for our model we formally develop the theorem that the optimized SMR has better performance than the single-round strategy. The following notions are used in the theorem.

- $\mathcal{E}^m$: the completion time for optimized SMR.
- $\mathcal{E}^s$: the completion time for Single-round strategy.
- T_I^m: the total idle time due to waiting for data transmission for optimized SMR.
- T_I^s: the total idle time due to waiting for data transmission for Single-round strategy.

Lemma 1. *The total idle time due to waiting for data transmission for optimized SMR is less than single round algorithm. That is, $T_I^m < T_I^s$.*

Theorem 1. *By assigning all N computing nodes to a task, the optimized SMR results in a shorter execution time than the single round algorithm. That is, $\mathcal{E}^m < \mathcal{E}^s$.*

5 Performance Evaluation

In this section, we evaluate the proposed two sets of real-time scheduling algorithms: Set 1 = {FIFO-MN-SMR and FIFO-MN-SMR*} and Set 2 = {EDF-MN-SMR and EDF-MN-SMR*}.

5.1 Simulation Configurations

We use a discrete simulator to simulate a collections of homogeneous clusters that are compliant with the system model presented in Section 3. Three parameters, N, τ and χ are specified for every cluster.

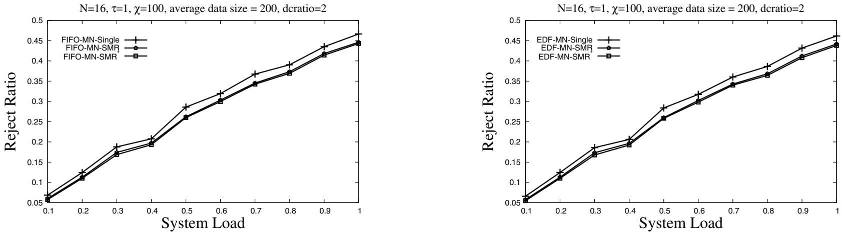
A task T_i is represented by (A_i, σ_i, D_i), where A_i, the task arrival time, is specified by assuming that the interarrival times follow an exponential distribution with a mean of $1/\lambda$, task data sizes σ_i are assumed to be normally distributed with the mean and the standard deviation equal to $Avg\sigma$, and task relative deadlines (D_i) are assumed to be uniformly distributed in the range $[\frac{AvgD}{2}, \frac{3AvgD}{2}]$, where $AvgD$ is the mean relative deadline. To specify $AvgD$, we use the term $DCRatio$ [4]. It is defined as the ratio of the mean deadline to the mean minimum execution time (cost), that is $\frac{AvgD}{\mathcal{E}(Avg\sigma,N)}$, where $\mathcal{E}(Avg\sigma, N)$ is the execution time assuming the task has an average data size $Avg\sigma$ and is allocated to run on all N nodes simultaneously [4]. Given a $DCRatio$, the cluster size N and the average data size $Avg\sigma$, $AvgD$ is implicitly specified as $DCRatio \times \mathcal{E}(Avg\sigma, N)$. Thus, task relative deadlines are related to the average task execution time. In addition, a task relative deadline D_i is chosen to be larger than its minimum execution time $\mathcal{E}(\sigma_i, N)$. In summary, we could specify the following parameters for a simulation: (N, τ, χ, $1/\lambda$, $Avg\sigma$, $DCRatio$).

We use the metric $SystemLoad$ [4] to analyze the cluster load for a simulation. It is defined as, $SystemLoad = \frac{\mathcal{E}(Avg\sigma,N)}{\lambda}$, which is the same as, $SystemLoad = \frac{TotalTaskNumber \times \mathcal{E}(N,Avg\sigma)}{TotalSimulationTime}$. To evaluate the performance of the real-time scheduling algorithms, we use the metric, *Task Reject Ratio*, defined as the ratio of the number of task rejections to the number of task arrivals [4]. The smaller the *Task Reject Ratio*, the better the real-time scheduling algorithm.

For all figures in this paper, a point on a curve corresponds to the average performance of ten simulations. For all ten runs, the same parameters (N, τ, χ, $SystemLoad$, $Avg\sigma$, $DCRatio$) are specified but different random numbers are generated for task arrival times A_i, data sizes σ_i, and deadlines D_i. For each simulation, the $TotalSimulationTime$ is 10,000,000 time units, which is sufficiently long.

5.2 Advantages of SMR

As proved in Section 4.3, for a single task, applying SMR will result in better performance. In our model, there are multiple dynamic tasks in the system. It is still unknown whether integrating SMR will give better performance than integrating the single round algorithm. Thus, in this section we compare two sets of real-time scheduling algorithms: Set 1 = {FIFO-MN-SMR and FIFO-MN-SMR*} and Set 2 = {EDF-MN-SMR and EDF-MN-SMR*} with algorithms FIFO-MN-Single and EDF-MN-Single, respectively [4]. We conducted simulations under the baseline system configuration $N = 16$, $\tau = 1$, $\chi = 100$, $Avg\sigma = 200$, *DCRatio*$= 2$.



(a) FIFO Algorithms **(b)** EDF Algorithms

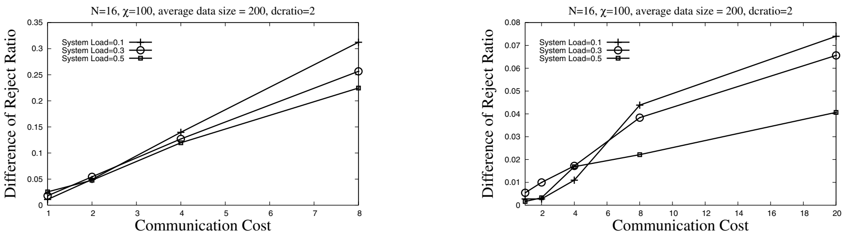
Fig. 3. Advantages of Integrating SMR

Figures 3a and 3b respectively show the comparison of algorithms FIFO-MN-SMR and EDF-MN-SMR, FIFO-MN-SMR* and EDF-MN-SMR* with their corresponding single-round algorithms FIFO-MN-Single and EDF-MN-Single. From Figure 3a we observe that FIFO-MN-SMR has a lower *Task Reject Ratio* than FIFO-MN-Single, which shows that applying SMR leads to better performance. We also observe that FIFO-MN-SMR* has even better performance. Figure 3b shows similar results. We conclude that it is very beneficial to integrate SMR with real-time divisible load scheduling.

5.3 Effects of Communication Cost (τ) on SMR

As discussed in [2], for multi-round algorithms, overlapping the data transmissions to different nodes are the key to shorten the execution time. Thus, the cost of communication has become an important issues that affects the performance. For this reason, we investigate the effect of communication cost on the performance of our approach.

For the simulation, we varied the values of τ from 1 to 2, 4 and 8, while keeping the other parameters constant as the baseline configuration. Each point in Figure 4a represents the difference in *Task Reject Ratios* for EDF-MN-Single and EDF-MN-SMR* algorithms. The three curves represent the cases when *SystemLoad* is equal to 0.1, 0.3 and 0.5 respectively. We observe that, as the



(a) EDF-MN-SMR* v.s. EDF-MN-Single

(b) EDF-MN-SMR* v.s. EDF-MN-SMR

Fig. 4. Effects of Communication Cost

communication cost increases, the difference in *Task Reject Ratios* becomes larger. This indicates that applying SMR has significant impact on the system performance as the communication cost increases. The reason is that compared to single round algorithms, the multi-round algorithms can pipeline the data transmission among different nodes, resulting in saving more communication time. Thus, when communication cost is high, algorithms integrating SMR will show more advantages.

We now compare the algorithm SMR with its optimized version. As discussed in Section 4.2, for the unoptimized SMR, the computing nodes finish at different times. Thus, when τ is large, that is, when the communication cost is high, the first node will finish computation much earlier than the last node. The optimized SMR fixes this problem by optimizing the partition of the last round. We now try to verify this effect by simulation.

We varied the values of τ from 1 to 2, 4, 8 and 20, while keeping the other parameters constant as the baseline configuration . Each point in Figure 4b represents the difference in *Task Reject Ratios* for EDF-MN-SMR and EDF-MN-SMR* algorithms. The three curves represent the cases when $SystemLoad$ is equal to 0.1, 0.3 and 0.5 respectively. We can observe that, as the communication cost increases, the difference in *Task Reject Ratios* becomes larger. Thus, when the communication cost is higher, the optimized SMR performs better. This verifies the conclusion above.

6 Conclusion

In this paper, we extend multi-round DLT to provide deterministic QoS to arbitrarily divisible applications executing in a cluster. SMR (Simplified Multi-Round) strategy is integrated into our previous algorithm framework [4]. We show that algorithms that integrate SMR perform better than the single-round algorithms in our model. We also investigate the effects of the communication costs on these algorithms. Experimental results show that when the communication cost is increasing, the algorithms integrating SMR have better performance. We theoretically prove that by assigning same number of nodes to a task, the optimized SMR will result in a shorter completion time than the single round approach.

References

1. Compact Muon Solenoid (CMS) Experiment for the Large Hadron Collider at CERN (European Laboratory for Particle Physics): Cms web page, `http://cmsinfo.cern.ch/Welcome.html/`
2. Bharadwaj, V., Robertazzi, T.G., Ghose, D.: Scheduling Divisible Loads in Parallel and Distributed Systems. IEEE Computer Society Press, Los Alamitos (1996)
3. Veeravalli, B., Ghose, D., Robertazzi, T.G.: Divisible load theory: A new paradigm for load scheduling in distributed systems. Cluster Computing 6(1), 7–17 (2003)
4. Lin, X., Lu, Y., Deogun, J., Goddard, S.: Real-time divisible load scheduling for cluster computing. In: 13th IEEE Real-Time and Embedded Technology and Application Symposium, Bellevue, WA, pp. 303–314 (2007)
5. Lin, X., Lu, Y., Deogun, J., Goddard, S.: Real-time divisible load scheduling with different processor available times. In: International Conference on Parallel Processing, Xian, China (2007)
6. Lin, X., Lu, Y., Deogun, J., Goddard, S.: Enhanced real-time divisible load scheduling with different processor available times. In: International Conference on High Performance Computing (2007)
7. Yang, Y., Casanova, H.: Umr: A multi-round algorithm for scheduling divisible workloads. In: Proceeding of International Parallel and Distributed Processing Symposium, p. 24 (2003)
8. Yeo, C.S., Buyya, R.: A taxonomy of market-based resource management systems for utility-driven cluster computing. Software: Practice and Experience (accepted, September 2005)
9. Amir, Y., Awerbuch, B., Barak, A., Borgstrom, R.S., Keren, A.: An opportunity cost approach for job assignment in a scalable computing cluster. IEEE Transactions on Parallel and Distributed Systems 11(7), 760–768 (2000)
10. Robertazzi, T.G.: Ten reasons to use divisible load theory. Computer 36(5), 63–68 (2003)
11. Kim, S., Weissman, J.B.: A genetic algorithm based approach for scheduling decomposable data grid applications. In: Proc. of International Conference on Parallel Processing (ICCP 2004), Montreal, Quebec, Canada, pp. 406–413 (2004)
12. Yu, D., Robertazzi, T.G.: Divisible load scheduling for grid computing. In: Proc. of IASTED International Conference on Paralle and Distributed Computing and Systems (PDCS 2003), Los Angeles, CA, USA (2003)
13. van der Raadt, K., Yang, Y., Casanova, H.: Practical divisible load scheduling on grid platforms with apst-dv. In: Proc. of 19th International Parallel and Distributed Processing Symposium (IPDPS 2005), Denver, CA, USA (2005)
14. Yang, Y., Casanova, H.: Multi-round algorithms for scheduling divisible workloads. IEEE Transactions on Parallel and Distributed Systems (TPDS) 16(11), 1092–1102 (2005)
15. Dertouzos, M.L., Mok, A.K.: Multiprocessor online scheduling of hard-real-time tasks. IEEE Trans. Softw. Eng. 15(12), 1497–1506 (1989)
16. Manimaran, G., Murthy, C.S.R.: An efficient dynamic scheduling algorithm for multiprocessor real-time systems. IEEE Trans. on Parallel and Distributed Systems 9(3), 312–319 (1998)

Energy-Efficient Dynamic Scheduling on Parallel Machines

Jaeyeon Kang and Sanjay Ranka

Department of Computer and Information Science and Engineering, University of Florida
{jkang,ranka}@cise.ufl.edu

Abstract. Energy consumption is a critical issue in parallel and distributed systems. Workflows consist of a number of tasks that need to be executed to complete an application. These tasks typically have precedence relationships that have to be observed during execution for correctness. DAGs (Directed Acyclic Graphs) can be used to represent many such workflows. The static algorithms to schedule for energy minimization under the deadline constraints are based on estimating worst case execution time for each task to guarantee that the application completes by a given deadline. During execution, many tasks may complete earlier than expected during the actual execution. This allows for adjusting the schedule for the tasks that have not yet begun execution to incorporate the extra slack. This has to be done with the dual goal of reducing the energy requirements while still meeting the deadline constraints. In this paper, we present a novel dynamic algorithm for remapping tasks for energy efficient scheduling of DAG based applications for DVS enabled systems. Our experimental results show that the combination of our dynamic assignment and dynamic slack allocation leads to significantly better energy minimization compared to not changing the static schedule and/or only performing dynamic slack allocation. Furthermore, its execution time requirements are small enough to be useful for a large number of applications.

1 Introduction

Computers use a significant and growing portion of the energy consumption in US. A study by Dataquest [10] reported that the world-wide total power dissipation of processors in PCs was 160MW in 1992, and by 2001 it had grown to 9000MW. Energy-aware computing is crucial for large-scale systems that consume considerable amount of energy and embedded systems that utilize battery for their power. Most effective energy minimization techniques are based on Dynamic Voltage Scaling (DVS). The DVS technique assigns differential voltages to each task to minimize energy requirements of an application. Assigning differential voltages is the same as allocating additional time or slack to a task. This technique has been found to be a very effective method for reducing energy in DVS enabled processors.

Workflows consist of a number of tasks that need to be executed to complete an application. These tasks typically have precedence relationships that have to be observed during execution for correctness. DAGs (Directed Acyclic Graphs) can be

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 208–219, 2008.

used to represent many such workflows. A DAG consists of nodes that represent computations or tasks and edges that represent the dependency between the nodes. DAGs have been shown to be representative of a large number of applications. A number of algorithms have been designed to schedule DAGs on parallel machines for energy minimization while meeting deadline. The following two step process is generally used for scheduling tasks with the goal of energy minimization while still meeting the deadline constraints:

1. Assignment: This step determines the mapping of tasks to processors and the ordering to execute tasks within a processor. Note that the finish time of DAG at the maximum voltage has to be less than or equal to the deadline for any feasible schedule.
2. Slack allocation: Once the assignment of each task is known, this step allocates variable amount of slack to each task so that the total energy consumption is minimized while the DAG can execute within a given deadline.

Most static algorithms for energy minimization developed in the literature for parallel and distributed machines use algorithms that minimize the total time requirements (the term makespan is also used for this purpose) during the assignment phase. Slack allocation algorithms are then used to minimize energy while still ensuring that the deadline constraints are met. The deadline is assumed to be longer than the makespan for obvious reasons. We have shown that incorporating energy requirements of tasks during the assignment process can lead to better overall energy minimization for homogeneous and heterogeneous processors. For heterogeneous machines, the energy requirements for a given task may be substantially different for each processor. We have shown that our algorithms can exploit the differential energy profiles effectively [6].

Worst-case execution time is used to guarantee that an application completes in a given time bound when the static scheduling is applied. In practice, many tasks may complete earlier than expected during the actual execution. This allows for other unexecuted tasks to potentially start earlier than what was envisioned during the static scheduling. This extra available slack can then be allocated to the tasks that have not yet begun execution such that the total energy requirements are reduced while still meeting the deadline constraints. Several runtime approaches for slack allocation have been studied in the literatures for independent tasks [1, 3, 8]. For DAG based workflows, a few dynamic scheduling algorithms have been recently proposed [4, 7]. These methods are based on allocating the slack generated (due to a task completing earlier than expected) at runtime. We have shown that reallocating slack at runtime (i.e., dynamic slack allocation) leads to better energy minimization [4]. We also showed that applying our dynamic slack allocation methods not only outperform the existing greedy method but also are comparable to static near optimal methods applied at runtime in terms of energy [4].

In this paper, we explore whether reassignment of tasks along with reallocation of slack during runtime can lead to even better performance in terms of energy minimization. For an approach that is effective at runtime, its overhead should be small for it to be useful. We present a novel dynamic scheduling algorithm that leads to good performance in terms of both computational time (i.e., runtime overhead) and energy requirements. The main features of our algorithm are as follows:

1. A small subset of tasks is chosen for reassignment to reduce the energy requirements.
2. The reassignment step is followed by a dynamic slack allocation step.
3. The original deadline constraints are met.

Our experimental results show that the combination of our dynamic assignment along with dynamic slack allocation leads to significantly better performance in terms of energy compared to (a) not changing the static schedule and (b) not changing the static mapping and only performing dynamic slack allocation. Furthermore, the time requirements are small enough that it should be useful for a large number of application workflows.

The remainder of this paper is organized as follows. In Section 2, we describe the energy model, the application model, and the overview of DVS scheme and assignment scheme. Section 3 presents our proposed dynamic scheduling algorithm for energy minimization. In Section 4, we present the performance of our algorithm through the experiment results. Section 5 provides the conclusion.

2 Preliminaries

Energy Model. Dynamic voltage scaling (DVS) technique reduces the dynamic power dissipation by dynamically scaling the supply voltage and thes clock frequency of processors. The relationship between power dissipation P_d, supply voltage V_{dd}, and frequency f is represented by $P_d = C_{ef} \cdot V_{dd}^2 \cdot f$ and $f = k \cdot (V_{dd} - V_t)^2 / V_{dd}$, where C_{ef} is the switched capacitance, k is the constant of circuit, and V_t is the threshold voltage [2]. The energy consumed to execute task τ_i, E_i, is expressed by $E_i = C_{ef} \cdot V_{dd}^2 \cdot c_i$, where c_i is the number of cycles required to execute the task. The supply voltage can be reduced leading to decreasing the processor speed and the energy consumption.

Application Model. Directed Acyclic Graph (DAG) represents the workflow among tasks. Fig. 1 (b) depicts the assignment for the DAG of Fig. 1 (a). The assignment is various depending on mapping methods while it satisfies a given deadline of the DAG. We assume that the deadline is larger than or equal to the finish time of the DAG (i.e., completion time of the DAG based on the assignment). Fig. 1 (c) represent an assignment DAG, which is the direct workflow among tasks generated after the assignment. The direct precedence relationship of tasks may change from one in an original DAG depending on the given assignment. For instance, tasks τ_1 and τ_4 have a direct dependency in the original DAG, but, in the assignment DAG, they have no direct dependency.

Slack Allocation Scheme. The slack allocation scheme (i.e. DVS scheme) allocates slack to tasks to minimize energy. The problem of slack allocation can be posed as the following: *Allocate variable amount of slack to each task so that the total energy consumption is minimized while the deadlines can still be met.* A Linear Programming (LP) based approach performs near optimal slack allocation has been described in [9].

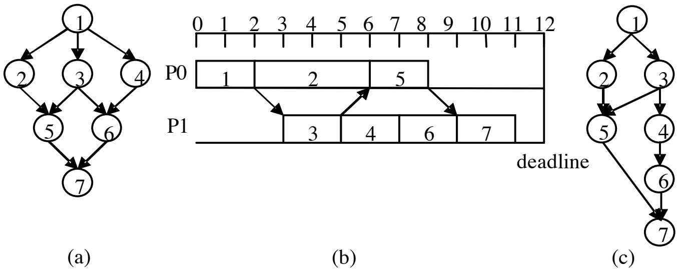


Fig. 1. (a) DAG, (b) Assignment on two processors, (c) Assignment DAG

Recently, we have developed another method for near optimal slack allocation that requires significantly lower computational time requirements [5]. It is worth noting that both the approaches have similar performance from the energy minimization perspective.

The path based algorithm in [5] (i.e., `PathDVS`) is an iterative approach that allocates a small amount of slack (called unit slack) in each iteration and finds a solution to the following problem: Find the subset of tasks that can be allocated this unit slack so that the total energy consumption is minimized while the deadline constraint is also met. This process is applied iteratively till all the slack is used. The dependency relationships in an assignment DAG constrain the total slack which can be allocated to different tasks. Each iteration of the problem can be reduced to finding a weighted maximal independent set of tasks, where the weight is given by the amount of energy reduction by allocating unit slack. The characteristic that each task is allocated the entire unit slack or no slack during each iteration allows for the use of search techniques to find the optimal slack allocation.

Assignment Scheme to Minimize DVS based Energy. The assignment determines the ordering to execute tasks and the mapping of tasks to processors based on the computation time at the maximum voltage level. Most of prior research on scheduling for energy minimization has not concentrated on the assignment process. Simple list based assignment algorithm with the goal of minimizing finish time were used for this purpose. However, we have recently shown that incorporating energy requirements of tasks during the assignment process can lead to significantly better overall energy minimization as compared to other existing algorithms [6]. The main feature of our assignment algorithm [6] is to consider the energy requirements based on potential slack during the assignment step. In other words, the algorithm assigns an appropriate processor for each task such that the total energy expected after slack allocation (i.e., expected DVS based energy) is minimized. The goal of the assignment is to minimize the total expected energy while still satisfying deadline constraints. Consider a scenario where the assignment of a subset of tasks has already been completed and a given next task in the prioritization list has to be assigned. The choice of the processors that can be assigned to this task should be limited to the ones where expected finish time from the overall assignment will lead to meeting the deadline constraints (else this will result in an infeasible assignment). Clearly, there is no guarantee that the schedule derived will be a feasible schedule (i.e., a schedule meeting deadline) at the time when the assignment for a given task is being determined because the feasibility of the schedule depends on the assignment of the other remaining tasks whose assignment is not determined. There are two main steps involved in the process of assigning a task to a processor [6]. These are briefly described below:

- *Estimating the Deadline for a given task*: The algorithm calculates the estimated deadline for each task, that is, deadline expected to enable a feasible schedule if the task can finish within its estimated deadline. The estimated deadline of a task is set to the latest finish time in order to allow more flexibility for processor assignment as the task can take a longer time to complete (while the probability of feasible schedule for DAG may be lower due to the higher probability of the increase of finish time). Here the latest finish time of a task is different based on its potential assigned processor due to the assignment-based dependency relationship among tasks. Thus, the time limit, which a task should be finished within, will vary for processors.
- *Processor Selection*: The task is assigned to a processor such that the total expected DVS based energy for the tasks that have already been assigned so far (and including the new task that is being considered for assignment) is minimized while trying to meet estimated deadline of the task. The candidate processors for the task are selected such that the task can execute by its estimated deadline. Note that the estimated deadline of a task may be different based on processors. Also, the processor selection process for the task depends on the number of candidate processors.

Details of each of the above steps are provided in [6]. It is worth noting the proposed dynamic scheduling algorithm is relatively independent of the static assignment scheme as the main benefits are to limit the actual tasks that are considered for applying the assignment process during runtime.

3 Dynamic Scheduling

We assume that a static scheduling algorithm has already been applied before executing tasks and the schedule needs to be adjusted whenever a task finishes before its scheduled time. Thus this schedule is updated whenever a dynamic scheduling is applied. When a task finishes before its estimated time, two changes may occur for all the remaining tasks (i.e., tasks that have not yet executed). Its processor mapping may change along with the start time and finish time. Also, the amount of slack (time over minimum execution time for that processor based on executing the task at maximum voltage) may change. The proposed dynamic scheduling algorithm utilizes several threads to generate a schedule: (1) one set for reallocating slack while keeping the assignment in the current schedule and (2) another set for changing the assignment and then reallocating slack. Then a schedule providing the minimum energy consumption is selected. For the dynamic scheduling (i.e., rescheduling), there are two steps that need to be addressed:

1. *Select the subset of tasks for rescheduling*: The potentially rescheduled tasks via the dynamic scheduling algorithm are tasks which have not yet started when the

algorithm is applied. We assume that the voltage can be selected before a task starts executing. The dynamic scheduling is applied to the subset of tasks among the tasks. The tasks considered for rescheduling are limited in order to minimize the overhead of reassigning processors and reallocating the slack during runtime. Clearly, this should be done so that the other goal of energy reduction is also met simultaneously.

2. *Determine the time range for the selected tasks*: The time range of the selected tasks has to be changed as some of the tasks have completed earlier than expected. Based on the computation time in the schedule and assignment-based dependency relationships among tasks, we recompute the time range (i.e., earliest start time and latest finish time) where the selected tasks should be executed. The time range is defined differently for reassignment and slack reallocation – time range over processors for reassignment and time range for the selected tasks given an assignment for slack reallocation. However, the main concept is same as the selected tasks have to be reassigned and reallocated slack within this time range in order to meet deadline constraints.

At this stage our proposed reassignment algorithm and slack reallocation approach are applied to the subset of tasks within the time range as described above. The computational time (i.e., runtime overhead) is kept small due to the limited number of tasks selected for rescheduling. While several assignment methods can be applied using threads, we propose a reassignment method based on our method described in [6]. This incorporates the expected DVS based energy information during the reassignment process.

The computation time of each selected task is set to its estimated execution time used in the assignment algorithm (before any static slack allocation) for rescheduling. In other words, the slack that was allocated during the static scheme is ignored for reassignment and slack reallocation. This effectively ensures that maximum flexibility is available for rescheduling. Furthermore, this will, in general, lead to better energy requirements as considering the change of assignment-based dependency relationships among tasks from the early finished task.

3.1 Choosing a Subset of Tasks for Rescheduling

The proposed dynamic scheduling algorithm, *k* lookahead approach, is based on choosing a subset of tasks for which the schedule will be readjusted [4]. The schedule for the remaining tasks (i.e., tasks not selected for the rescheduling) is not affected. Using *k* lookahead approach, all tasks within a limited range of time are considered for the readjustment of schedule. The range of time is limited with to *k * maximum computation time of any task*. The set of tasks selected for the rescheduling when task τ_l finishes early is defined by

$$\Gamma_{allocation} = \{\tau_i \mid staticSTime_i \geq ftime_l,\ staticFTime_i \leq ftime_l + k * \max_{\tau_j \in \Gamma} compTime_j\},$$
$$\text{where } \tau_l \text{ s.t. } ftime_l \neq staticFTime_l \tag{1}$$

where $staticSTime_i$ is the start time of task τ_i in the static or previous schedule, $staticFTime_i$ is the finish time of task τ_i in the static or previous schedule, $ftime_l$ is thes actual finish time of task τ_l at runtime, and $compTime_j$ is the computation time of task

τ_j on its assigned processor, a.k.a., the estimated execution time at the maximum voltage.

The approach with '*all*' option for *k* (i.e., *k-all* lookahead approach) corresponds to the static scheduling approach without the limitation on the time range for tasks considered for rescheduling. Thus, the *k-all* lookahead approach is same as applying the static scheduling algorithm to all the remaining tasks at runtime. One would expect this to be close to the best that can be achieved. The set of tasks selected for the rescheduling when task τ_l finishes early is defined by

$$\Gamma_{allocation} = \{\tau_i \mid staticSTime_i \geq ftime_l\}, \text{where } \tau_l \text{ s.t. } ftime_l \neq staticFTime_l \qquad (2)$$

3.2 Time Range for Selected Tasks

The schedule for tasks not in the set of reschedulable tasks is kept to be the same (this is based on static schedule or schedule generated by last rescheduling). For the set of reschedulable tasks, a range of time to execute each task is defined based on the current feasible solution (before applying the dynamic scheduling algorithm). The time range is differently defined for reassignment and slack reallocation. For reassignment, the time range is defined for each processor as the task may be mapped to more than one processor. And, for slack reallocation, the time range is defined for a particular processor.

For reassignment, the time range of processors is computed as follows:

1. The available start time of each processor is the possible earliest start time of each processor for the tasks. It is set to the expected finish time (i.e., the finish time in the current schedule) of the last task that is not in the set of reschedulable tasks and already started when applying an algorithm (currently executing or already finished) on each processor (i.e., a task with the latest finish time on each processor among tasks not in the set of reschedulable tasks). It is worth noting that it is not the earliest start time of reschedulable tasks on each processor. The earliest start time of the tasks on a processor is different due to the precedence relationships among other tasks. The available start time of a processor p_j, *procSTime*$_j$, is defined by

$$\begin{aligned} &procSTime_j = staticFTime_i, \\ &where\ proc_i = p_j\ \&\ staticSTime_i < ftime_l\ \&\ \max_i staticSTime_i \end{aligned} \qquad (3)$$

2. The deadline of each processor is the possible latest finish time of each processor for the tasks. It is set to the expected start time (i.e., the start time in the current schedule) of the first task that is not in the set of reschedulable tasks and is not started yet when applying an algorithm on each processor (i.e., a task with the earliest start time on each processor among tasks not in the set of reschedulable tasks). It is worth noting that it is not the latest finish time of reschedulable tasks on each processor. Like the earliest start time, the latest finish times of the tasks on a processor are different due to the precedence relationships among other tasks. The deadline of a processor p_j, *procDeadline*$_j$, is defined by

$$procDeadline_j = staticSTime_i,\quad where\ \ \tau_i \in \Gamma^c_{allocation}\ \&\ proc_i = p_j\ \&\ \min_i staticSTime_i \tag{4}$$

Given an assignment, for slack reallocation, the time range is defined as follows [4]:

1. The start time of the tasks is changed as flexibly as possible to meet the deadline constraints as well as the finish times of assignment-based predecessors of each task. In particular, the finish time of the predecessors that have already completed or are not part of the selected tasks is fixed.
2. The finish time (or deadlines) of the tasks is changed so that they can be completed as late as possible while ensuring that the deadline constraints are met.

Using the above constraints, each reschedulable task has different amount of the maximum available slack. The actual slack is computed to be within the time range for reschedulable tasks. The maximum available slack for each task is used while computing the estimated energy for reassignment and reallocating slack after reassignment. The maximum available slack of reschedulable task τ_i, $slack_i$, is defined by the difference of the latest start time of τ_i, LST_i, and the earliest start time of slack reallocable task τ_i, EST_i. These values are computed as follows:

$$LST_i = \min\left(deadline_i, LST_{pSucc_i}, \min_{\tau_j \in succ_i} \left(LST_j - commTime_{ij} \right) \right) - staticCTime_i \tag{5}$$

$$EST_i = \max\left(\begin{array}{l} start_i, \left(EST_{pPred_i} + staticCTime_{pPred_i} \right) \\ \max\limits_{\tau_j \in pred_i} \left(EST_j + staticCTime_j + commTime_{ij} \right) \end{array} \right) \tag{6}$$

$$slack_i = LST_i - EST_i \tag{7}$$

where $deadline_i$ is the deadline of task τ_i, $staticCTime_i$ is the computation time of task τ_i in the current schedule, $commTime_{ij}$ is the communication time between task τ_i and task τ_j on their assigned processors, $start_i$ is the start time of task τ_i, $succ_i$ is the set of direct successors of task τ_i in a DAG, $pSucc_i$ is the task assigned next to task τ_i on the same assigned processor, $pred_i$ is the set of direct predecessors of task τ_i in a DAG, and $pPred_i$ is the task assigned prior to task τ_i on the same assigned processor. Here the earliest start time and the latest start time of a task not included in the set of slack reallocable tasks are equal to its static start time or its actual start time at runtime if it already finished (i.e., $EST_j = LST_j = staticSTime_j,\ where\ \ \tau_j \in \Gamma^c_{allocation}$). Please note that the static start time of a finished task is same to the actual start time.

4 Experimental Results

In this section, we compare the performance of the combination of dynamic assignment and dynamic slack allocation proposed in this paper (i.e., `DynamicAssgn`) with the following two main methods which outperform other existing in each given state:

- Static scheduling (i.e., `StaticDVS`) [5]: This static scheduling provides near optimal solutions for energy minimization given an assignment. However, it keeps the schedule generated at compile time during runtime.
- Dynamic slack allocation (i.e., `DynamicDVS`) [4]: This dynamic slack allocation readjusts the schedule whenever a task finishes earlier than expected during runtime while keeping a given assignment. In our experiments, *k-3* time lookahead slack allocation approach which gives good performance in terms of energy is used.

The dynamic algorithms (i.e., `DynamicDVS`, `DynamicAssgn`) are applied to a static schedule that is based on a known assignment algorithm and a static slack allocation algorithm. We use a static scheduling algorithm presented in [5]. Also, for a fair comparison with `DynamicDVS`, *k-3* time lookahead approach for `DynamicAssgn` is used and `PathDVS` [5] is used as a slack allocation method applied at runtime.

DAG Generation: We randomly generated a large number of graphs with 50 and 100 tasks. The execution time of each task on each processor at the maximum voltage is varied from 10 to 40 units and the communication time between a task and its child task for a pair of processors is varied from 1 to 4 units. The energy consumed to execute each task on each processor is varied from 10 to 80. The execution of graphs is performed on 4, 8, 16, and 32 processors.

Dynamic Environments Generation: There are two broad parameters for dynamic environments: *earlyFinishedTaskRate* and *timeDecreaseRate*.

- The number of tasks that finish earlier than expected in the schedule is given by the *earlyFinishedTaskRate* (i.e., *number of early finished tasks = earlyFinishedTaskRate * total number of tasks*).
- The amount of decrease for each task that finishes early is given by *timeDecreaseRate* (i.e., *amount of decrease = timeDecreaseRate * estimated execution time*).

We experimented with *earlyFinishedTaskRate*s equal to 0.2, 0.4, 0.6, and 0.8 and *timeDecreaseRate*s equal to 0.1, 0.2, 0.3, and 0.4. In this paper, only the results for *earlyFinishedTaskRates* equal to 0.2 and 0.8 are presented due to space limitations.

Performance measures: The deadline is defined by *deadline = (1 + deadline extension rate) * total finish time from assignments without DVS scheme*. We experimented with deadline extension rates equal to 0 (no extension), 0.01, 0.02, 0.05, 0.1, and 0.2, but only the results for no deadline extension are presented due to space limitations. To compare algorithms, the normalized energy consumption, that is, total energy normalized by the energy obtained from the static assignment (before applying static slack allocation), is used. The computational time (i.e., runtime overhead) is also performed as an important measure for algorithms in dynamic environments.

Fig. 2 shows the comparison of our algorithm with static scheduling and dynamic slack allocation in terms of energy consumption with respect to different time decrease rates and different early finished task rates for 4, 8, 16, and 32 processors. Based on the results, the combination of dynamic assignment and dynamic slack

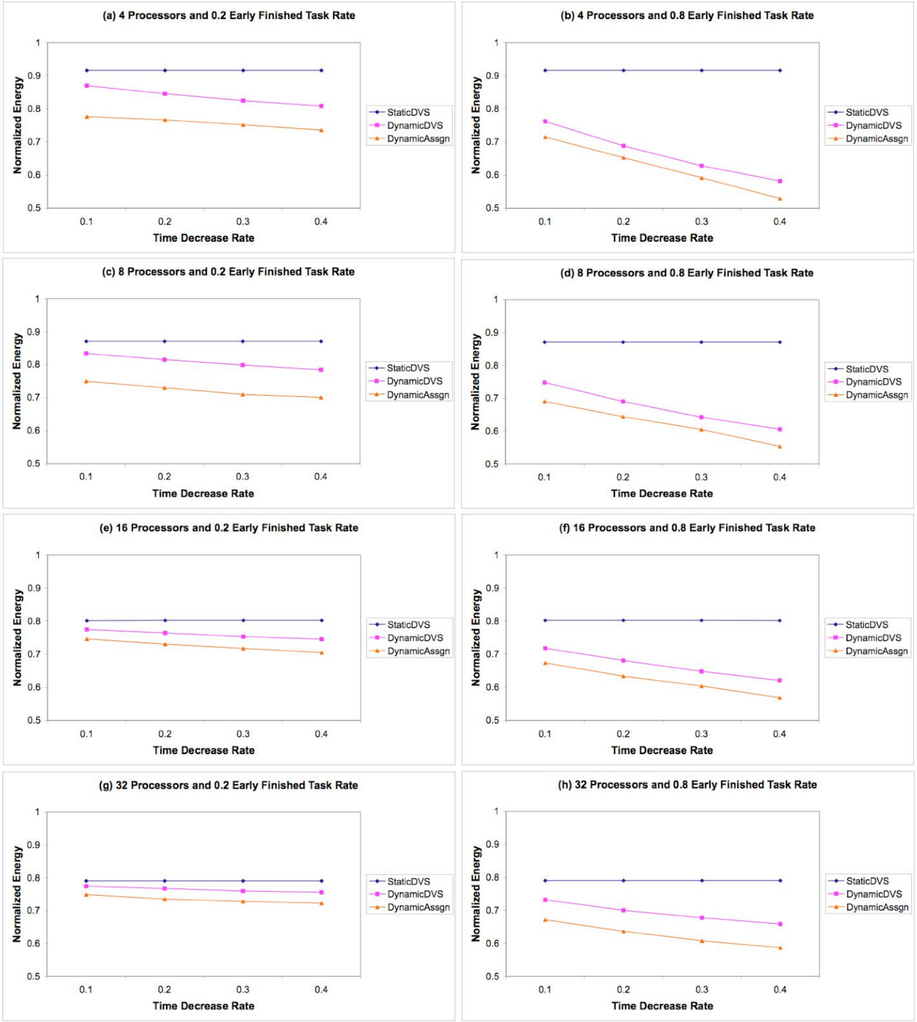


Fig. 2. Normalized energy consumption for 50 and 100 tasks on different number of processors

allocation (i.e., `DynamicAssgn`) significantly outperforms static scheduling (i.e., `StaticDVS`) and dynamic slack allocation (i.e., `DynamicDVS`) in terms of energy consumption. For instance, for 32 processors, `DynamicAssgn` improves energy requirements by 15-26% and 8-12% compared to `StaticDVS` and `DynamicDVS` respectively. These results show that both adjusting the assignment as well as adjusting the slack is necessary for minimizing the energy requirements. Furthermore, the improvement of `DynamicAssgn` over the other two algorithms increases as the *timeDecreaseRate* increases. Note that the performance of `StaticDVS` is independent of the *timeDecreaseRate* because it does not change the generated schedule during runtime.

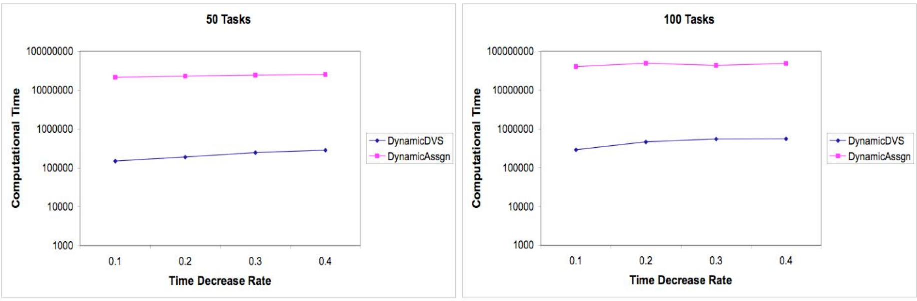


Fig. 3. Computational time to readjust the schedule from an early finished task with respect to different time decrease rates (unit: ns – via logarithmic scale)

Fig. 3 shows the average time requirement to readjust the schedule due to a single task's early finish (i.e., runtime overhead). The computational time of DynamicAssgn is two orders of magnitude larger than DynamicDVS since DynamicAssgn requires assignment process as well as slack allocation process. However, DynamicAssgn requires 0.02-0.04 seconds in average to readjust the schedule at runtime – this small time should make it applicable for a large number of compute intensive applications.

5 Conclusion

We have presented a novel scheduling algorithm to minimize energy consumption for dynamic environments, where the actual execution time of a task may be different from its estimated time. We showed that our algorithm (i.e., the combination of reassignment and slack reallocation) provides considerably better energy minimization compared to static scheduling. It also provides significant improvements over only reallocating the slack at runtime without changing the assignment.

Acknowledgements. This work is supported in part by the National Science Foundation under Grant ITR 0325459 and 0312038 (under a subcontract from FIU). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Aydin, H., Melhem, R., Mossé, D., Mejía-Alvarez, P.: Power-Aware Scheduling for Periodic Real-Time Tasks. IEEE Trans. on Computers 53(5), 584–600 (2004)
2. Chandrakasan, A.P., Sheng, S., Brodersen, R.W.: Low-Power CMOS Digital Design. IEEE J. of Solid-State Circuits 27(4), 473–484 (1992)
3. Jejurikar, R., Gupta, R.: Dynamic Slack Reclamation with Procrastination Scheduling in Real-Time Embedded Systems. In: Jejurikar, R., Gupta, R. (eds.) Design Automation Conf., pp. 111–116 (2005)

4. Kang, J., Ranka, S.: Dynamic Algorithms for Energy Minimization on Parallel Machines. In: Euromicro Conf. on Parallel, Distributed and Network-Based Processing, pp. 399–406 (2008)
5. Kang, J., Ranka, S.: DVS based Energy Minimization Algorithm for Parallel Machines. IEEE Int. Parallel and Distributed Processing Sym., 1–12 (2008)
6. Kang, J., Ranka, S.: Assignment Algorithm for Energy Minimization on Parallel Machines, University of Florida Technical Report (2008)
7. Mishra, R., Rastogi, N., Zhu, D., Mossé, D., Melhem, R.: Energy Aware Scheduling for Distributed Real-Time Systems. In: Int. Parallel and Distributed Processing Sym., p. 21b (2003)
8. Shin, Y., Choi, K.: Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In: Design Automation Conf., pp. 134–139 (1999)
9. Zhang, Y., Sharon Hu, X., Chen, D.Z.: Task Scheduling and Voltage Selection for Energy Minimization. In: Design Automation Conf., pp. 183–188 (2002)
10. Dataquest, `http://data1.cde.ca.gov/dataquest/`

A Service-Oriented Priority-Based Resource Scheduling Scheme for Virtualized Utility Computing

Ying Song[1,2,3], Yaqiong Li[1,2,3], Hui Wang[1,2], Yufang Zhang[1,2,3], Binquan Feng[1,2,3], Hongyong Zang[1,2,3], and Yuzhong Sun[1,2]

[1] Key Laboratory of Computer System and Architecture,
Institute of Computing Technology, Chinese Academy of Sciences, China
[2] National Research Center for Intelligent Computing Systems, ICT, China
[3] Graduate University of Chinese Academy of Sciences, China
songying@ncic.ac.cn

Abstract. In order to provide high resource utilization and QoS assurance in utility computing hosting concurrently various services, this paper proposes a service computing framework-RAINBOW for VM(Virtual Machine)-based utility computing. In RAINBOW, we present a priority-based resource scheduling scheme including resource flowing algorithms (RFaVM) to optimize resource allocations amongst services. The principle of RFaVM is preferentially ensuring performance of some critical services by degrading of others to some extent when resource competition arises. Based on our prototype, we evaluate RAINBOW and RFaVM. The experimental results show that RAINBOW without RFaVM provides 28%~324% improvements in service performance, and 26% higher the average CPU utilization than traditional service computing framework (TSF) in typical enterprise environment. RAINBOW with RFaVM further improves performance by 25%~42% for those critical services while only introducing up to 7% performance degradation to others, with 2%~8% more improvements in resource utilization than RAINBOW without RFaVM.

Keywords: Resource scheduling, utility computing, virtualization.

1 Introduction

It's a new trend towards providing heterogeneous services concurrently by enterprise data centers, for example, of Google and Amazon. Google provides services consisting of Google search, Google office, and Youtube. In the past, those services were provided by different platforms. In such a case, QoS guaranteeing of services with the time-varying capacities (including computing, storage, and communication) demands as the result of request arrival distributions resulted in over-provision each service. Such datacenters were often underutilized. One approach to increase the resource utilization is consolidating services in a shared infrastructure-utility computing [2]. In such a shared platform, isolation among the hosted services is crucial. As virtualization is increasingly popular, utility computing is incorporating virtualization technology such as virtual machines (VMs) with effective isolation among services. Many companies envisioning

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 220–231, 2008.

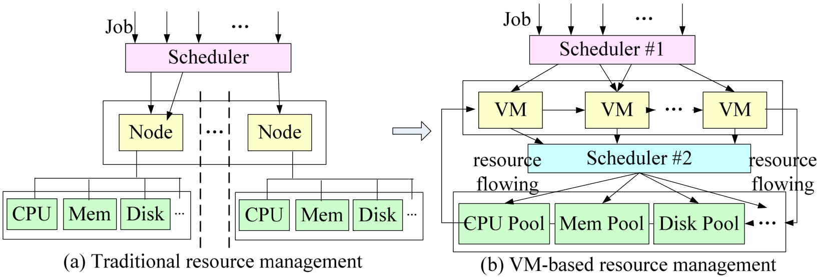


Fig. 1. The evolution of resource management

this popular trend have devoted themselves to developing new utility computing infrastructures based on virtualization technologies such as VMware [17] and XenSource [11].

VM-based utility computing has some obvious advantages like consolidation, isolation, flexibility resource management. Dynamic load changes of services give rise to dynamic capacities demands, which implies it is necessary to control ***resource flowing*** among services on-demand. Resource flowing denotes the process in which resources released by VMs are allocated to others. The VM-based resource management differs from the previous works in the granularity (from nodes to components) and dimensions (from one to two), illustrated in fig.1. Traditional resource management corresponds to the scheduler in fig.1(a), which dispatches jobs onto a set of exclusively servers. As to the VM-based resource management (fig.1(b)), scheduler #1 corresponding to the traditional resource management dispatches jobs onto a set of VMs. It adds a new dimensioned resource scheduler (scheduler #2) to optimize the usage of finer-grained resources via resource flowing among VMs. Any contemporary VMMs (Virtual Machine Monitor, i.e. Xen and VMware) with resource reallocating scheme provide technical support rather than strategy to resource flowing. They need better scheduler#2 to optimize the usage of resources. Optimizing resource flowing among VMs is a challenge in such a platform.

In order to address above challenge, we propose a novel service computing framework --RAINBOW for VM-based utility computing. RAINBOW integrates separate resources into a shared virtual computing platform, ensuring QoS and higher resource utilization. We model the resource flowing using optimization theory. Based on this model, we present a priority-based resource scheduling scheme including a set of algorithms of resource flowing among VMs (RFaVM). The principle of RFaVM is preferentially ensuring performance of some critical services by degrading of others to some extent when resource competition arises. We implement a Xen-based prototype to evaluate RAINBOW and RFaVM. We consider CPU and memory flowing schemes which could be generally applicable to other resources. The experimental results indicate that RAINBOW with RFaVM effectively improves the resource utilizations, and meets QoS goals of services, in the typical enterprise IT environment with inappreciable overheads.

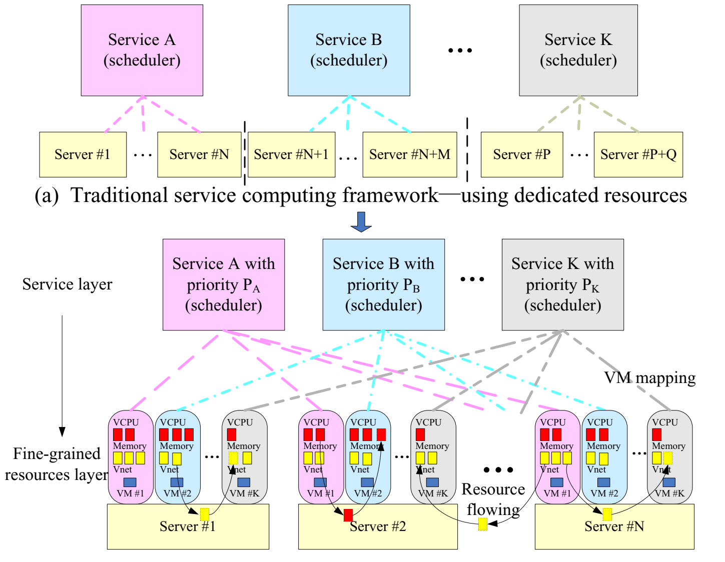
This paper has the following contributions. 1) We propose a novel service computing framework (RAINBOW) for VM-based utility computing. 2) We model the resource flowing in RAINBOW and present a priority-based resource scheduling scheme including a set of algorithms of resource flowing among VMs (RFaVM).

The rest of this paper is organized as follows: RAINBOW is introduced in Section 2. Section 3 describes resource scheduling scheme, while Section 4 discusses the experimental results. Section 5 presents related work. Section 6 concludes the paper.

2 A Novel Service Computing Framework – RAINBOW

2.1 RAINBOW Statement

We present a novel service computing framework (RAINBOW, illustrated in fig.2(b)) to improve the resource utilization and QoS. Different from the traditional service computing framework (TSF, illustrated in fig.2(a)) in which one service runs on a set of dedicated servers, RAINBOW uses virtualization to isolate concurrent services in a shared physical infrastructure. We observe that diverse services may be various resource-bound, for example, VoD service is I/O-bound, HPC service is CPU-bound. Further, we obtain another observation that diverse services may have various time-varying resource demands as the result of request arrival distributions [4][13]. Those two observations motivate our design of RAINBOW. In order to minimize the interaction among the hosted services due to their competitions for resources, services with the same resource-bound should be distributed onto different physical servers. In RAINBOW, a set of VMs serving a particular service is called a ***group***. The key principle is that VMs belonging to a single group may be split onto multiple physical



(a) Traditional service computing framework—using dedicated resources

(b) RAINBOW—VM-based service computing framework for utility computing

Fig. 2. The evolution of service computing framework

servers, while each server hosts VMs belonging to different groups. Each service dispatches workloads to VMs in its group according to its scheduling algorithms. RAINBOW provides resources to the hosted services on-demand via resource flowing taking the priority of service into account. This allows better resource utilization and QoS at the service level compared to previous proposals [12] in scenarios where there is competition for the same resource by similar service components.

Resource flowing strategy should solve four problems. 1) Which resource will flow? 2) When will such resource flow? 3) Which VMs will be the source and target of flow? 4) How many resources will flow? In order to answer these four problems, we model the resource flowing in RAINBOW first.

2.2 Resource Flowing Model

We consider the problem of resource flowing among VMs in RAINBOW, and model it by optimization theory. This model is a general one which can be respectively used by CPU, memory or other resources. Based on the model, we present a set of algorithms of resource flowing amongst VMs (RFaVM) to provide resources to the hosted services on-demand. First we introduce the following notations and concepts:

- R - The total CPU, memory or other resources in a server.
- K - The number of VMs resided in the server.
- $C_{i\text{-}min}$ - The minimum threshold of resources allocated to VM$_i$, which is used to avoid huge interaction among VMs when competition for resources arises. It is set by experience in our experiments, and will be justified in the near future.
- R_{it} - Resources allocated to VM$_i$ at time t. It obeys: $R \geq \sum_{i=1}^{K} R_{it}$ and $R_{it} \geq C_{i\text{-}min} > 0$.
- D_{it} - Resources demands by VM$_i$ at time t. It is proportional to the requests arrival rate.
- SP_i - The static priority of service hosted in VM$_i$. It indicates how critical the requirement for QoS. If $i>j$, $SP_i \geq SP_j$. SP_i is determined by administrator.
- Φ_i - The tolerable QoS threshold of service hosted in VM$_i$.
- Q_{it} - Quality of service hosted in VM$_i$ at time t. The smaller the Q_{it} is, the better QoS the service gains. As we all know QoS (i.e. response time) is decided by the required and allocated resources, namely, $Q_{it}=f_i(R_{it},D_{it})$. The relationships can be found by studying the typical services, which is our ongoing work. In order to fairly weight various services using their QoS, we use the rate of QoS of service hosted in VM$_i$ at time t and of the tolerable QoS (Q_{it}/Φ_i, ***QoS-rate*** for short).

The goal of resource flowing is to optimize QoS taking the priority into account, giving the limited resources. It is an optimization problem with limiting conditions. Thus, we select the programming model of optimization theory to model the resource flowing. In order to provide resource flowing with a utility function, which maps QoS of the target to a benefit value, we define the utility function UF_t[1]. UF_t is related to the static priorities and *QoS-rates* of the hosted services. The resource flowing is how to control resource allocation among VMs with the goal of minimizing the utility function UF_t, giving the limited resources, formulated as formula (1).

We simplify the above model to design our resource flowing algorithms. All the functions $f_i(R_{it},D_{it})/\Phi_i$ are set as: if $D_{it}>R_{it}$, $f_i(R_{it},D_{it})/\Phi_i=D_{it}-R_{it}$; else, $f_i(R_{it},D_{it})/\Phi_i=0$. Applying the Simplex Method, we get the resolution of this simplified model. If $D \leq R$,

R_{it}=D_{it}; else, we give priority to allocating resources to VMs with higher priority. The relationship between D_{it} and R_{it} can be directly reflected by the resource utilization of VM$_i$. Thus, our algorithms control resource flowing according to the resource utilization of each VM and static priority of each service.

$$\min \quad UF_t = \sum_{i=1}^{K} \frac{Q_{it}}{\Phi_i} \times SP_i = \sum_{i=1}^{K} \frac{f_i(R_{it}, D_{it})}{\Phi_i} \times SP_i \tag{1}$$

$$s.t. \begin{cases} \sum_{i=1}^{K} R_{it} \le R \\ R_{it} \ge C_{i-\min} \qquad (i = 1,2,\cdots,K) \end{cases} .$$

3 Priority-Based Resource Scheduling Scheme

We consider CPU and memory flowing algorithms. We assume that there are K VMs hosting various services in a server. The efficacy of our algorithms is intimately dependent on how well it can predict resource utilization. We take a simple low-overhead last-value-like prediction as reference [15] does, in which the resource utilization during the last interval is used as a predictor in the next interval.

3.1 CPU Flowing Amongst VMs Algorithm (CFaVM)

The hypervisor (Xen) we used provides a credit scheduler that can operate in two modes: capped and non-capped. We select the non-capped mode to prevent from the degradation of CPU utilization. In the non-capped mode, if multiple VMs require idle processing units, the scheduler distributes idle processing units to the VMs in proportion to their weights. There is no support on automatically changing weights of VMs according to its workloads and QoS. CFaVM uses a hybrid priority scheme to adjust the dynamic priority (weight) of each VM according to its static priority and resource utilization. Some VMs with high priority are preferentially guaranteed with the rapidly increased dynamic priorities, while others suffer performance degradations via the slowly increased dynamic priorities when competition for CPU arises.

W refers to the weight assigned to a VM, *OW* refers to the weight of a VM in the prior interval, $\triangle W$ means the increased weight when a VM is CPU overload. We fix W_1 to avoid frequent changes of W_i when CPU competitions arise. The changes of other W_i alter the CPU share of VM$_1$. We define $\triangle W$ to be in proportion to *SP*. Formula (2) gives the relationship of $\triangle W_i$ and $\triangle W_j$. We define the minimum CPU resources allocated to VM_i ($C_{i\text{-}min}$), which could be denoted by the maximum ($W_{i\text{-}max}$) and minimum ($W_{i\text{-}min}$) weight thresholds assigned to VM$_i$. We define $W_{j\text{-}max}$ to be in proportion to SP_j as formula (3).

$$\triangle W_i = (SP_i/SP_j) * \triangle W_j. \tag{2}$$

$$W_{j\text{-}max} = (SP_j/SP_j) * W_{j\text{-}max}. \tag{3}$$

```
Input: $U_i$, $OW_i$, i=1,…,K; Output: $W_i$, i=1,…,K;
Algorithm:
Initialize $OW_i$ (i=1,…,K) to be 256. $SP_i$, $\triangle W_i$, $W_{i\text{-max}}$, $W_{i\text{-min}}$ (i=1,…,K) are initialized.
While (1)
{     For (i=1,i<=K,i++)
      {
          1) If ($U_i$=100%) and ($OW_i+\triangle W_i<W_{i\text{-max}}$)
             Then $W_i=OW_i+\triangle W_i$;
             Else if ($U_i>=T_u$) and ($OW_i<W_{i\text{-max}}$) and($OW_i+\triangle W_i>=W_{i\text{-max}}$)
                 Then $W_i=W_{i\text{-max}}$;
          2) If $U_i<T_d$  Then {  $W_i=\frac{OW_i\times U_i\times(\sum_{j=1}^{K}OW_j-OW_i)}{T_d\times(\sum_{j=1}^{K}OW_j)-OW_i\times U_i}$
                  If $W_i<W_{i\text{-min}}$
                  Then $W_i=W_{i\text{-min}}$;
                  }
      } sleep(1);
} End;
```

Fig. 3. CFaVM

Based on the CPU utilization of each VM, CFaVM (fig.3) determines whether the CPU is overload in a VM or not. We choose T_u as the threshold of CPU overload and T_d as the desired CPU utilization. If the CPU utilization of VM$_i$ reaches T_u, CPU resources should flow in VM$_i$ (we call it a ***consumer VM***). CFaVM increases $\triangle W_i$ on the weight of VM$_i$ to increase CPU resources. Other VMs will be the ***provider VMs***, which flow out CPU in proportion to their weights. If the CPU utilization of VM$_j$ is lower than T_d, CPU should flow out from VM$_j$. The weight of VM$_j$ is decreased to decrease CPU resources. In order to let the CPU utilization of VM$_j$ reach T_d, the new weight of VM$_j$ is calculated as formula (4).

$$\because \frac{OW_j}{\sum_{i=1}^{K}OW_i}\times R\times U_j=\frac{W_j}{W_j+\sum_{i=1}^{K}OW_i-OW_j}\times R\times T_d \quad \therefore W_j=\frac{OW_j\times U_j\times(\sum_{i=1}^{K}OW_i-OW_j)}{T_d\times(\sum_{i=1}^{K}OW_i)-OW_j\times U_j}. \tag{4}$$

3.2 Lazy Memory Flowing amongst VMs Algorithm (LMFaVM)

In LMFaVM, M refers to the total memory which could be used by the K VMs. Each VM is initially allocated M/K memory. The minimum memory threshold of VM$_i$ ($M_{i\text{-}min}$), corresponding to $C_{i\text{-}min}$ in section 2.2, is defined as formula (5).

$$M_{i-\min}=(SP_i/\sum_{j=1}^{K}SP_j)*M/K \tag{5}$$

$$\triangle M_i=(SP_i/SP_1)*\triangle M_1 \tag{6}$$

Based on the static priority and the collected idle memory of VM$_i$ (IM_i), LMFaVM (fig.4) determines whether there is memory overload in a VM or not. If idle memory of each VM is higher than the threshold Ψ, no memory flows. If IM_i is lower than Ψ, LMFaVM increases $\triangle M_i$ (illustrated as formula (6)) or less memory to VM$_i$ (consumer VM), as long as there is another VM (provider VM) which can give its memory to VM$_i$. A VM can be a provider when its memory is more than its minimum

```
Input: $IM_i$, $M_i$, i=1,…,K;   Output: $M_i$, i=1,…,K;
Algorithm:
Initialize $M_i$ (i=1,…,K) as M/K. $SP_i$,$\triangle M_i$ and $M_{i-min}$ (i=1,…,K)are initialized.
While (1)
{    For (i=1,i<=K,i++)
        {    If ($IM_i$<$\Psi$)
             {    1) select VMx with maximum E from VMs which could give their memory to $VM_i$;
                  2) control  min{$\triangle M_i$,$M_x$-$M_{x-min}$} memory flow from VMx  to $VM_i$;
             }
        }  sleep(T);
};  End;
```

Fig. 4. LMFaVM

memory threshold and either of the two conditions is satisfied: 1) it has idle memory; 2) its priority is lower than that of the consumer VM. We define *E* to be the rate of *IM* and *SP* (*E=IM/SP*) to select the provider VM. If memory of a VM is overload, LMFaVM compares *E* of VMs which can be the provider, and selects the VM with maximum *E* to be the final provider VM.

4 Implementation and Performance Evaluation

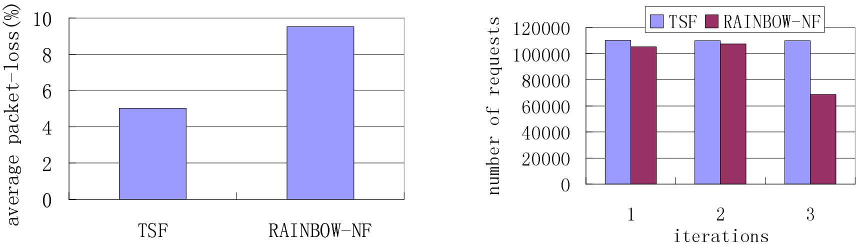
The implementation of our prototype is based on Xen. In our servers pool, there are four servers each of which has two 2190MHz Dual Core AMD Opteron(tm) processors with 1024KB of cache and 4GB of RAM. We use CentOS4.4, and Xen-3.0.4. We use other four machines to be clients of services. The systems are connected with a Gigabit Ethernet. Using our prototype, we do a set of experiments to evaluate RAINBOW and RFaVM. The experiments are classified into two groups: Group-I evaluates RAINBOW without resource flowing (‘***RAINBOW-NF***’ for short); Group-II verifies RFaVM in RAINBOW.

Group-I: We evaluate the strengths and weaknesses of our RAINBOW-NF using the following two comparisons with various scenarios.

Comparison-I: We compare RAINBOW-NF with TSF using the following three typical enterprise services: web, HPC, and office services. On each server, we create 3 VMs. Each VM is allocated 1 VCPU and 1GB memory. VMs devoted to a service are distributed onto the four servers. Apache [14], LVS [19] with Round Robin algorithm, and the e-commerce workloads of SPECWeb2005 [22] are used for the web server. We use Condor [16] to dispatch Linpack [20] jobs to the HPC VMs. The office service is provided by our office applications virtualization product. It separates the display from the running of applications based on virtual desktop, and it distributes applications to servers according to workloads in these servers. We use Xnee [3] to emulate the real-world office applications based on the trace collected by [6].

As to the office service, we collect the starting up time of eight applications (FTP, mozilla, openoffice, etc.) for 4 times as the performance metric which is captured by the modified VNC. We compare the average starting up time of all the applications. The number of requests with good response (respond within 3 seconds) defined by SPECWeb, is the performance metric of the web service. The HPC service is evaluated by throughput (the number of completed linpack jobs during 3 hours). The experiment results show that RAINBOW-NF provides dramatic improvements both in service performance (28%~324%) and in the CPU utilization (26%) over TSF.

Comparison-II: We evaluate RAINBOW-NF using two I/O-bound services (VoD and web). Helix [21] is used for the VoD server and we make the VoD workloads generator according to ref[4]. Figure 5(a)(b) show the comparisons of TSF and RAINBOW-NF by the two services. From these figures we can see that service performance is degrading by RAINBOW-NF, especially for the web service in its third iteration when the VoD service reaches its peak of workloads.



(a) The VoD service

(b) The web service

Fig. 5. Comparisons between TSF and RAINBOW

Analysis and Conclusion on Group-I: In Comparison-I, since the three services are different resource-bound, their resource bottlenecks are different. Distributing the same service onto multiple physical servers reduces the demands on its bottleneck resources on each server. On the other hand, each server hosts concurrently multiple services with diverse resource-bound, which avoids frequent competitions for the same resource. Thus, RAINBOW-NF provides huge improvements. In Comparison-II, the two services are I/O-bound. Frequent competitions for I/O, as well as Xen's I/O overhead, leads to huge interactive impacts between services in RAINBOW.

Group-II: We verify RFaVM using the same experimental scenario and benchmarks as in the Comparison-I of Group-I. On each physical server, we create 3 VMs. We initially allocate various resources to VMs (table 1) to provide different conditions (BN denotes the bottleneck resources when services reach their peaks of workloads).

Table 1. Initial resources allocation to VMs on various conditions

BN	CPU	memory
CPU	1 VCPU (VCPUs of the 3 VMs running on the same server are pinned to the same physical core)	1G
memory	1 VCPU (mapping to all the physical cores)	600M
CPU& mem	1 VCPU (the same as BN=CPU)	600M

The baseline system RAINBOW-NF provides static resources allocation to VMs. 'CFaVM' refers to RAINBOW adopting CPU flowing algorithm. 'LMFaVM' uses memory flowing algorithm in RAINBOW. 'RFaVM' adds CFaVM and LMFaVM.

SP_{office}:SP_{web}:SP_{hpc} is initialized as 4:3:1. We choose 50MB idle memory as the threshold of memory overload, 90% as the threshold of CPU overload (T_u), and 70% as the desired CPU utilization (T_d) for VMs. These parameters are decided by our experience. Table 2 illustrates the comparisons results on various conditions (***A*:*B*** denotes that *A* compares with *B*). From these results we can see that RFaVM provides great improvements (42%) in performance of the office service and tiny improvements (2%) in performance of the web service, while introducing small impairments (up to 7%) in performance of the HPC service. This is the result that RFaVM preferentially ensures performance of the office service by degrading of others to some extent, and it gives preference to the web service when the web service competes for resources with the HPC service.

Table 2. Various scheduling algorithms in RAINBOW vs. RAINBOW-NF ('NF' for short)

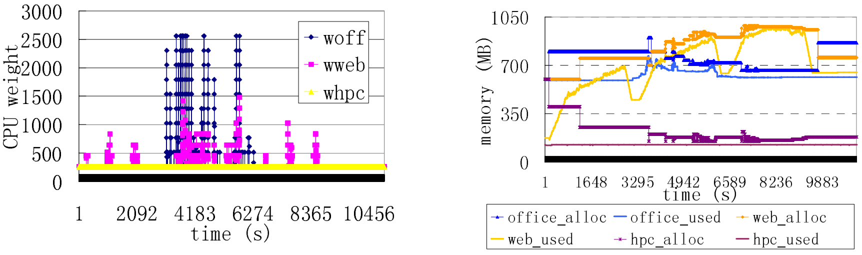
BN	Comparisons	Office	web	HPC	Resource utility
CPU	CFaVM: NF	25%	1%	-7%	CPU:2 %
mem	LMFaVM: NF	37%	1%	1%	mem:8%
CPU&mem	RFaVM: NF	42%	2%	1%	CPU:2%; mem:6%

We compare our resource flowing in RAINBOW ('RAINBOW' for short) with ref[12] in table 3. Ref[12] only focuses on CPU reallocation, while RAINBOW focuses on both CPU and memory flowing. The working intervals of RAINBOW are 1s for CPU and 5s for memory, which are smaller than that of ref[12] (10s). This implies that RAINBOW has faster response to the change of resource requirements by services. We compute the improvement and degradation introduced by ref[12] using its fig.14-15 of response time with and without their controller. The improvement provided by ref[12] (28%) is smaller than that provided by RAINBOW (42%) for the critical service. The degradation introduced by ref[12] (41%) is much larger than that introduced by RAINBOW (7%) for other services. These results imply that our scheme is better than ref[12] in the aspect of assuring QoS.

Table 3. RAINBOW vs. ref[12]

	resources	working interval	Improvement	Degradation
ref[12]	CPU	10s	28%	41%
RAINBOW	CPU&mem	1s(CPU), 5s(mem)	42%	7%

We use CPU flowing as an example of analyzing the reason why RFaVM further improves performance of the critical services by slightly impairing of others using fig.6(a)-(b) and fig.7. Figure 6(a)-(b) illustrate the behaviors of adjusting CPU weights and memory of VMs controlled by RFaVM. In fig.6(a), *woff* denotes weight of office VM, *wweb* and *whpc* are similar. Figure 6(b) illustrates memory allocated to services (denoted by service_alloc) adopting RFaVM and memory used by services (denoted by service_used). Figure 7 shows the CPU requirement by workloads of each service. From fig.7 we can see that in the first 3000 seconds (3000s for short) and after 7000s, competitions for CPU arise between web and HPC services. The



(a) CPU weights of VMs

(b) Memory owned&used by VMs

Fig. 6. Behaviors of resources flowing using RFaVM

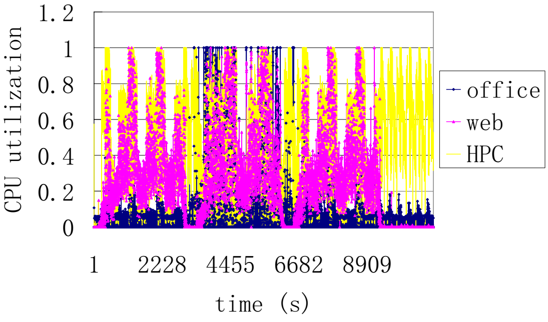


Fig. 7. CPU utilization of VMs

priority of web service is higher than that of HPC service. RFaVM controls CPU flow to the web VM (fig.6(a)). From 3000s to 7000s, office service with the highest priority requires more CPU than in other periods. Thus RFaVM controls CPU flow to the office VM when CPU competitions arise, resulting in more CPU allocated to the office VM in this period than in other periods (fig.6(a)). The priority of HPC service is the lowest one. RFaVM controls CPU flow out from the HPC VM when CPU competitions arise, which results in less CPU allocated to the HPC VM than others (fig.6(a)). As the result, CPU flowing improves the performance of office and web services by impairing that of HPC service.

In our experiments, the interval of resource flowing ranges from 1s to 1118s for CPU, and ranges from 5s to 2555s for memory. Our scheme leads to the overhead of collecting information and controlling resource flow. Collecting information leads to 8M memory overhead. Each resource flow leads to 0%~0.3% CPU overhead. Such overhead can be ignored since it is inappreciable to the system.

5 Related Work

Currently, a large body of papers is on managing data center in a utility computing environment that provides on-demand resources, such as HP's SoftUDC [9], IBM's

PLM [5], and VMware DRS [17]. To the best of our knowledge, no other studies proposed the same service computing framework and scheduling algorithms as ours.

Several studies provide on-demand resources at the granularity of servers in the data center. Oceano [8] dynamically reallocates whole servers for an e-business computing utility. SoftUDC [9] proposes a software-based utility data center. It adopts the strategy of on-the-fly VM migration, which is also implemented by VMware's VMotion [10], to provide load balancing. Ref[18] dynamically allocates resources to applications via adding/removing VMs on physical servers. They are in contrast to our scheme that controls resource allocation at the granularity of CPU time slots and memory.

There is a growing body of work on providing on-demand fine-grained resources in the shared data center. IBM's PLM [5] and VMware DRS [17] dynamically allocate resources to partitions/VMs according to shares specified statically and resource utilization, ignoring QoS. In [7], a two-level resource management system with local controllers at the VM level and a global controller at the server level is proposed. Ref[12] dynamically adjusts CPU shares to individual tiers of multi-tier applications in the VM-based data center. They choose the design where one server hosts multiple VMs which provide the same service. This is different from our RAINBOW that puts various services hosted in VMs onto the same physical server and distributes the same service onto different physical servers. Ref[12] uses the CPU capped mode to provide performance isolation between VMs, which decreases QoS and resource utilization. Based on the resource flowing model, our scheme focuses on both CPU and memory flowing and uses the CPU non-capped mode to achieve better QoS than [12].

6 Conclusion

This paper presents a novel service computing framework (RAINBOW) for VM-based utility computing. In RAINBOW, a priority-based resource scheduling scheme including a set of algorithms of resource flowing amongst VMs (RFaVM) is proposed according to the time-varying resources demands and QoS goals of various services. We implement a Xen-based prototype to evaluate RAINBOW and RFaVM. The results indicate that RAINBOW without RFaVM gains improvements in the typical enterprise IT environment (improves 28%~324% in service performance, as well as 26% in CPU utilization over TSF), while introducing degradation in the case of hosting multiple I/O-bound services, which results from competitions for I/O and Xen's I/O overhead. RFaVM further improves performance by 25%~42% for those critical services while only introducing up to 7% performance impairment to others, with 2%~8% improvements in average physical resource utilization than RAINBOW without RFaVM. Compared with ref[12], our resource scheduling is better in assuring QoS. Such results indicate that RFaVM gains its goals of service differentiation as well as improvements in resource utilization.

Some inviting challenges remain in this research. The parameters in our algorithms will be justified in the near future. In order to ensure RAINBOW work smoothly, admission control in RAINBOW is still open.

Acknowledgments. This work was supported in part by the National High-Tech Research and Development Program (863) of China under grants 2006AA01Z109, 2006AA01A102, and 2007AA01Z119, China's National Basic Research and Development 973 Program (2006CB303106), and the projects of Natural Science Foundation of China (NSFC) under grants 90412013.

References

[1] Menascé, D.A., Bennani, M.N.: Autonomic Virtualized Environments. In: ICAS 2006, p. 28 (2006)

[2] Hewlett-Packard: HP Utility Data Centre – Technical White Paper (October 2001), http://www.hp.com

[3] Sandklef, H.: Testing applications with Xnee. Linux Journal 2004(117), 5 (2004)

[4] Yu, H., Zheng, D., Zhao, B.Y., et al.: Understanding User Behavior in Large-Scale Video-on-Demand Systems. In: EuroSys 2006, pp. 333–344 (2006)

[5] IBM Redbook: Advanced POWER Virtualization on IBM System p5: Introduction and Configuration (January 2007)

[6] Wang, J., Sun, Y., Fan, J.: Analysis on Resource Utilization Patterns of Office Computer. In: PDCS 2005, pp. 626–631 (2005)

[7] Xu, J., Zhao, M., et al.: On the Use of Fuzzy Modeling in Virtualized Data Center Management. In: ICAC 2007, p. 25 (2007)

[8] Appleby, K., et al.: Oceano-SLA based management of a computing utility. In: Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management, pp. 855–868 (2001)

[9] Kallahalla, M., Uysal, M., Swaminathan, R., et al.: SoftUDC: A Software-Based Data Center for Utility Computing, November 2004, pp. 38–46. IEEE Computer Society, Los Alamitos (2004)

[10] Nelson, M., Lim, B.H., Hutchins, G.: Fast Transparent Migration for Virutal Machines. In: 2005 USENIX Annual Technical Conference, pp. 391–394 (2005)

[11] Barham, P., Dragovic, B., et al.: Xen and the art of virtualization. In: SOSP, pp. 164–177 (2003)

[12] Padala, P., Zhu, X., Uysal, M., et al.: Adaptive Control of Virtualized Resources in Utility Computing Environments. In: EuroSys 2007, pp. 289–302 (2007)

[13] Wang, Q., Makaroff, D.: Workload Characterization for an E-commerce Web Site. In: Proc. CASCON 2003, pp. 313–327 (2003)

[14] Fielding, R.T., Kaiser, G.: The Apache HTTP Server Project. IEEE Internet Computing 1(4), 88–90 (1997)

[15] Govindan, S., Nath, A.R., Das, A.: Xen and Co.: Communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In: VEE, pp. 126–136 (2007)

[16] Tannenbaum, T., Wright, D., Miller, K., et al.: Condor - A Distributed Job Scheduler. In: Sterling, T. (ed.) Beowulf Cluster Computing with Linux, pp. 307–350. The MIT Press, Cambridge (2002)

[17] VMware Infrastructure: Resource Management with VMware DRS

[18] Wang, X., Lan, D., Wang, G., et al.: Appliance-based Autonomic Provisioning Framework for Virtualized Outsourcing Data Center. In: ICAC 2007, p. 29 (2007)

[19] http://www.linuxvirtualserver.org/

[20] http://www.netlib.ort/benchmark/phl/

[21] http://www.realnetworks.com

[22] http://www.spec/org/web2005/

Scalable Processing of Spatial Alarms

Bhuvan Bamba[1,⋆], Ling Liu[1], Philip S. Yu[2,∗], Gong Zhang[1], and Myungcheol Doo[1]

[1] College of Computing, Georgia Institute of Technology
[2] Department of Computer Science, University of Illinois at Chicago
{bhuvan,lingliu,gzhang3,mcdoo}@cc.gatech.edu, psyu@cs.uic.com

Abstract. Spatial alarms can be modeled as location-based triggers which are fired whenever the subscriber enters the spatial region around the *location of interest* associated with the alarm. Alarm processing requires meeting two demanding objectives: high accuracy, which ensures zero or very low alarm misses, and system scalability, which requires highly efficient processing of spatial alarms. Existing techniques like periodic evaluation or continuous query-based approach, when applied to the spatial alarm processing problem, lead to unpredictable inaccuracy in alarm processing or unnecessarily high computational costs or both. In order to deal with these weaknesses, we introduce the concept of *safe period* to minimize the number of unnecessary spatial alarm evaluations, increasing the throughput and scalability of the server. Further, we develop alarm grouping techniques based on locality of the alarms and motion behavior of the mobile users, which reduce *safe period computation costs* at the server side. An evaluation of the scalability and accuracy of our approach using a road network simulator shows that the proposed approach offers significant performance enhancements for the alarm processing server.

1 Introduction

Time-based alarms are effective reminders of future events that have a definite time of occurrence associated with them. Just as time-based alarms are set to remind us of the arrival of a *future reference time point*, spatial alarms are set to remind us of the arrival of a *spatial location of interest*. Thus, spatial alarms can be modeled as location-based triggers which are fired whenever a mobile user enters the spatial region of the alarms. Spatial alarms provide critical capabilities for many location-based applications ranging from real time personal assistants, inventory tracking, to industrial safety warning systems.

A mobile user can define and install many spatial alarms; each alarm is typically shared by one or many other users. Alarms can be classified into three categories based on the publish-subscribe scope of the alarm as *private*, *shared* or *public* alarms. *Private* alarms are installed and subscribed to exclusively by the alarm owner. *Shared* alarms are installed by the alarm owner with a list of

[⋆] This work was performed while the author was at IBM T.J. Watson Research Center.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 232–244, 2008.

k ($k > 1$) authorized subscribers and the alarm owner is typically one of the subscribers. Mobile users may subscribe to *public* alarms by topic categories or keywords, such as *"traffic information on highway 85North"*, *"Top ranked local restaurants"*, to name a few. Each alarm is associated with an *alarm target* which specifies the location of interest to the user; a region surrounding the alarm target is defined as the *spatial alarm region.* The *alarm trigger* condition requires that subscribers of the alarm be notified as soon as they enter the spatial alarm region.

Processing of spatial alarms requires meeting two demanding objectives: high accuracy, which ensures no alarms are missed, and high scalability, which guarantees that alarm processing is efficient and scales to large number of spatial alarms and growing base of mobile users. The conventional approach to similar problems involves periodic evaluations at a high frequency. Each spatial alarm evaluation can be conducted by testing whether the user is entering the spatial region of the alarm. Though periodic evaluation is simple, it can be extremely inefficient due to frequent alarm evaluation and the high rate of irrelevant evaluations. This is especially true when the mobile user is traveling in a location that is distant from all her location triggers, or when all her alarms are set on spatial regions that are far apart from one another. Further, even a very high frequency of alarm evaluations may not guarantee that all alarms will be successfully triggered. The *spatial continuous query approach* would process a spatial alarm by transforming the alarm into a *user-centric* continuous spatial query. Given the alarm region of radius r around the alarm target and the mobile user's current location, the transformed spatial query is defined by the query range r with the mobile alarm subscriber as the focal object of the query. The query processor checks if the obtained query results contain the alarm target object. This process repeats periodically until the alarm target is included in the query results at some future time instant. The obvious drawback of this approach is the amount of unnecessary processing performed in terms of both the number of evaluations and the irrelevant query result computation at each evaluation. A more detailed discussion of the weaknesses can be found in our technical report [6].

Spatial alarms can be processed using server-based infrastructure or client-based architecture. A server-based approach must allow optimizations for processing spatial alarms installed by multiple mobile clients, whereas a client-based approach focuses more on energy-efficient solutions for evaluating a set of spatial alarms installed on a single client. Bearing in mind the problems inherent with the continuous spatial query evaluation approach and drawbacks of the periodic alarm evaluation approach, we develop a safe period-based alarm evaluation approach. The goal of applying safe period optimization is to minimize the amount of unnecessary alarm evaluations while ensuring zero or very low alarm miss rate. The other technical challenge behind safe period optimization is to minimize the amount of safe period computation, further improving system scalability and achieving higher throughput. We describe our basic approach for safe period computation in the next section and address the challenge of reducing the amount of safe period computations in Section 3. We evaluate the scalability and accuracy of our approach using a road network simulator and show that our proposed framework offers significant performance enhancements for the alarm processing server while maintaining high accuracy of spatial alarms.

2 Safe Period Computation

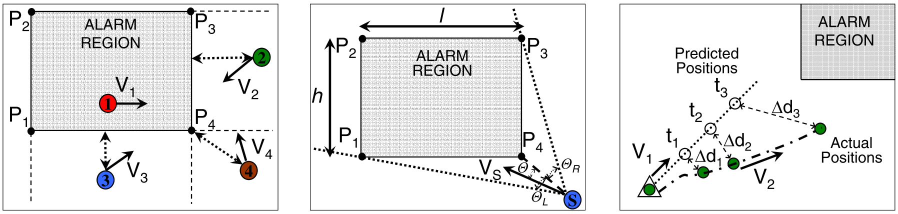
Safe period is defined as the duration of time for which the probability of an alarm being triggered for a subscriber is zero. Consider a subscriber S_i $(1 \leq i \leq N)$ and a spatial alarm A_j $(1 \leq j \leq M)$, where N is the total number of mobile users and M is the total number of alarms installed in the system. The safe period of alarm A_j with respect to subscriber S_i, denoted by $sp(S_i, A_j)$ can be computed based on the distance between the current position of S_i and the alarm region R_j, taking into account the motion characteristics of S_i and alarm target of A_j. Concretely, for alarms with mobile subscribers and static targets, the two factors that influence the computation of safe period $sp(S_i, A_j)$ are (i) the velocity-based motion characteristic of the subscriber S_i, denoted by $f(V_{S_i})$ and (ii) the distance from the current position of subscriber S_i to the spatial region R_j of alarm A_j, denoted by $d(S_i, R_j)$. Thus the safe period $sp(S_i, A_j)$ can be computed as follows:

$$sp(S_i, A_j) = \frac{d(S_i, R_j)}{f(V_{S_i})} \tag{1}$$

2.1 Distance Measurements

We use *Euclidean distance* as the basic distance measure for safe period computation. It measures the minimum distance from the current position of the mobile user, denoted as $P_m = (x_m, y_m)$, to the spatial alarm region R. Consider a spatial alarm region R covering the rectangular region represented by four vertices of a rectangle (P_1, P_2, P_3, P_4), as shown in Figure 1(a). The minimum Euclidean distance from P_m to the spatial alarm region R, denoted by $d_{m,R}$, can be computed by considering the following four scenarios: ① when the mobile subscriber lies inside the spatial alarm region the distance $d_{m,R}$ is zero; ② when the mobile subscriber is within the y scope of the spatial alarm region, the minimum euclidean distance is the distance from the mobile subscriber to the nearer of the two spatial alarm edges parallel to the x-axis; ③ when the mobile subscriber is within the x scope of the spatial alarm region, minimum euclidean distance is the distance from the mobile subscriber to the nearer of the two spatial alarm edges parallel to the y-axis; and ④ when the mobile subscriber is outside both the x and y scope then the distance is the minimum of the euclidean distance to the four vertexes. Formally, $d_{m,R}$, the minimum Euclidean distance from mobile position P_m to the spatial alarm region R, is computed using the following formula:

$$d_{m,R} = \begin{cases} 0, & x_1 \leq x_m \leq x_2 \\ & \mathit{and}\ \ y_1 \leq y_m \leq y_2 \\ min(|x_m - x_1|, |x_m - x_2|), & y_1 \leq y_m \leq y_2\ \mathit{only} \\ min(|y_m - y_1|, |y_m - y_2|), & x_1 \leq x_m \leq x_2\ \mathit{only} \\ min(d_{m,1}, d_{m,2}, d_{m,3}, d_{m,4}), & \mathit{otherwise} \end{cases}$$



(a) Euclidean Distance (b) Steady Motion (c) Dead Reckoning

Fig. 1. Basic Safe Period Computation

$d_{m,k}$, $k \in \{1, 2, 3, 4\}$ denotes the Euclidean distance from P_m to rectangle vertex P_k. The distance function $d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ is used to compute the Euclidean distance between two points P_i and P_j.

The safe period formula in equation 1 assumes that the subscriber heads towards the alarm region in a straight line along the direction of the minimum Euclidean distance, an assumption that rarely holds true. One way to relax this stringent condition is to use the *steady motion assumption*: If the subscriber is heading towards the alarm region R, then the deviation in her motion direction is not likely to be extreme. Figure 1(b) shows a scenario where the bounded deviation in subscriber motion is taken into account for calculating average safe period for subscriber S approaching alarm region R. In order for the subscriber S to enter the alarm region R at some future time instant, the average angle of motion for the subscriber S over the safe period must lie between $-\theta_L$ and $+\theta_R$ (as shown in the figure), which we refer to as *alarm trigger angular range.* Assume that the mobile subscriber heads towards the alarm region R in a direction at an angle θ to the minimum Euclidean distance vector; we refer to the distance from the subscriber position to the alarm region as the steady motion distance, denoted as $sm_{dist}(\theta)$. The steady motion-based safe period can be determined by $sm_{dist}(\theta)/f(V_S)$. Using the average steady motion distance obtained by computing $sm_{dist}(\theta)$ over all θ values ranging from $-\theta_L$ to $+\theta_R$, the steady motion-based safe period over the alarm trigger angular range can be calculated as,

$$sp = \frac{\int_{-\theta_L}^{+\theta_R} sm_{dist}(\theta) d\theta}{f(V_S) \int_{-\theta_L}^{+\theta_R} d\theta} = \frac{l + h}{f(V_S)(\theta_R + \theta_L)}, \tag{2}$$

where l, h denote the length and height of the spatial alarm region. The steady motion assumption provides a more realistic and optimistic measure for safe period computations compared to the minimum Euclidean distance approach.

2.2 Velocity Measurements

The use of maximum travel speed of the mobile client for the velocity function $f(V_S)$ carries both advantages and disadvantages. On one hand, the 'maximum travel speed' can be set by pre-configuration based on a number of factors, such as the nature of the mobile client (car on the move or a pedestrian walking on the street), or the type of road used. On the other hand, maximum speed-based estimation is often pessimistic, especially in the following two scenarios: (i) when the mobile client stops for an extended period of time, or (ii) when the mobile client suddenly turns onto a road with very low speed limit. Another issue related to the use of maximum speed of a mobile client for the velocity function $f(V_S)$ is related to alarm misses. The maximum velocity-based approach may fail to trigger alarms in cases where the maximum speed for the mobile subscriber increases suddenly. For example, a vehicle moving from a street onto a state highway would experience a sudden increase in its velocity, which may invalidate safe period computations. One way to address such sudden increase in velocity is to use *dead reckoning* techniques which require the mobile user to report to the server when her velocity increases over a certain threshold, as shown in Figure 1(c). The use of dead reckoning or similar techniques will allow the server to recompute the safe period for mobile client upon any significant velocity change. In Figure 1(c), the mobile client keeps track of its predicted positions based on its maximum speed and its actual positions. As soon as the difference between the predicted position and the actual position exceeds a given threshold value (say δ), the client provides its current speed to the server.

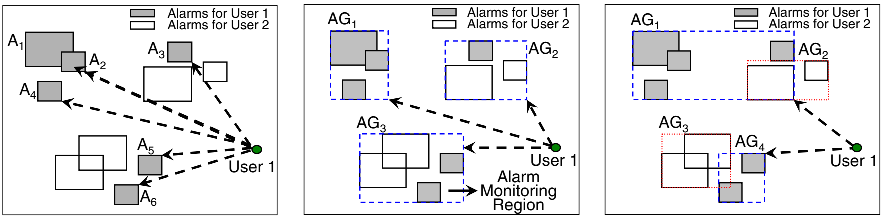
2.3 Safe Period-Based Alarm Evaluation

The safe period-based approach processes a spatial alarm in three stages. First, upon the installation of a spatial alarm, the safe period of the alarm with respect to each authorized subscriber is calculated. Second, for each alarm-subscriber pair, the alarm is processed upon the expiration of the associated safe period and a new safe period is computed. In the third stage, a decision is made regarding whether the alarm should be fired or wait for the new safe period to expire.

When compared to periodic alarm evaluation, the safe period approach for spatial alarm processing reduces the amount of unnecessary alarm evaluation steps, especially when the subscriber is far away from all her alarms. On the other hand, the main cost of the basic safe period approach described in this section is due to the excessive amount of unnecessary safe period computations, as the basic safe period approach performs safe period computation for each alarm-subscriber pair. One obvious idea to reduce the amount of unnecessary safe period computations is to group spatial alarms based on geographical proximity and calculate safe period for each subscriber and alarm group pair instead of each alarm-subscriber pair.

3 Alarm Grouping Techniques

The basic premise behind alarm grouping is to reduce the number of safe period computations while ensuring no alarm misses. In this section, we present three alternative grouping techniques, each of which offers different degree of improvement for safe period computations. First, we group all alarms based on



(a) Per Alarm Safe Period (b) SL Grouping (c) SSSL Grouping

Fig. 2. Alarm Locality-based Grouping

their spatial locality. Alternatively, we apply spatial locality based-grouping to alarms of each individual subscriber. Our experimental study shows that this approach is more effective. The third locality-based alternative is to employ the nearest alarms-based grouping, which is effective but costly when there are frequent alarm additions and removals.

3.1 Spatial Locality-Based Grouping

Spatial locality-based (SL) grouping considers the set of alarms from all users and groups together the nearby alarms. This approach outperforms basic safe period alarm evaluation if each group has a large number of alarms belonging to the same subscriber. Figure 2(a) displays the alarm regions for a set of installed alarms. The alarms for user 1 are marked by shaded regions. Basic safe period evaluation computes the distance from each of the six alarms $\{A_i \mid 1 \leq i \leq 6\}$. In comparison, Figure 2(b) shows three groups derived from spatial locality-based grouping technique. We use a simple R-tree implementation in order to group alarms and identify the *minimum bounding rectangles (MBRs)* for alarm groups which are also referred to as *alarm monitoring regions*. Instead of computing distance for each alarm-subscriber pair, spatial locality-based grouping calculates the distance for each subscriber and alarm group pair. However, on entering a monitoring region the distance to all relevant alarms within the alarm group also needs to be computed. Despite this additional evaluation step, the number of safe period computations may be considerably reduced by grouping alarms according to spatial locality. Instead of six computations required by the basic safe period technique, only three computations need to be performed as all three alarm groups, $\{AG_i \mid 1 \leq i \leq 3\}$, contain alarms relevant to user 1. Further computations are dependent on the number of relevant alarms within the users' current alarm monitoring region. Even though this approach reduces the number of computations it requires considerable additional processing to determine the set of relevant alarm groups for each subscriber and the set of relevant alarms for each subscriber within an alarm group. The lack of subscriber-specificity in the underlying data structure, R-Tree, leads to retrieval of large number of unnecessary alarms. This technique proves to be efficient in presence of large number of public alarms as the effect of subscriber-specificity is reduced in this situation.

3.2 Subscriber-Specific Spatial Locality-Based Grouping

In contrast to spatial locality-based grouping, subscriber-specific spatial locality-based (SSSL) grouping performs a two level grouping: the first level grouping is on all subscribers and the second level grouping is on spatial alarms relevant to each subscriber. We use a B-tree based implementation to speed up search on subscribers and an R-Tree implementation to capture spatial locality of alarms for each subscriber. The underlying data structure is a hybrid structure which uses a B-tree for subscriber search at the first level and an R-tree for subscriber specific spatial alarm search at the second level. Figure 2(c) shows an example of this grouping. Alarms installed by user 1 are grouped together in AG_1 and AG_4 and may be fired only when the user is entering the $MBRs$ of AG_1 or AG_4. Subscriber specific spatial locality-based grouping has two advantages over the previous approaches. First, the number of safe period computations is significantly reduced. Second, each alarm group contains alarms relevant to a single user, thus no irrelevant processing is performed. Our experimental results show that this approach is efficient in the presence of large number of subscribers and for large number of private and shared alarms.

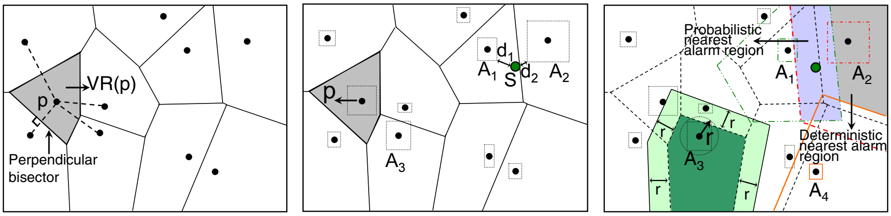
3.3 Nearest Alarms-Based Grouping

Nearest alarms-based grouping allows the system to perform one or only a few alarm checks dependent on the current subscriber position. The idea is to have each location on the map associated with the nearest spatial alarm region(s). In order to perform nearest alarms-based grouping we use an extension of the well known Voronoi diagram geometric structure [5]. The Voronoi diagram for a given set of points P in d-dimensional space $\mathbb{R}^d$ partitions the space into regions where each region includes all points with a common closest point ϵ P. The *Voronoi region VR(p)* corresponding to any point $p \ \epsilon \ P$ contains all points $p_i \ \epsilon \ \mathbb{R}^d$ such that,

$$\forall p' \epsilon P, p' \neq p, dist(p_i, p) \leq dist(p_i, p') \tag{3}$$

Figure 3(a) shows the Voronoi diagram for a set of points in two-dimensional space $\mathbb{R}^2$ with euclidean distance metric. The shaded area marks out the *Voronoi region VR(p)* for the point p.

In order to create a Voronoi diagram for spatial alarms we first represent each spatial alarm region R by its center point (x_{cr}, y_{cr}) and *l, h* representing the length and height of the alarm region. We consider the center point of each alarm region as a Voronoi site and create the Voronoi diagram as shown in Figure 3(b). But alarm regions may overlap with adjacent Voronoi regions as for alarm A_3 in the figure. Also, consider the subscriber S in the figure residing in the Voronoi region of alarm A_1. S is at a minimum Euclidean distance d_1 from the alarm region of A_1 and at a minimum Euclidean distance d_2 to the alarm region of A_2. Even though $d_2 < d_1$, A_1 is incorrectly identified as the nearest alarm on the basis of the underlying Voronoi diagram. In order to rectify this problem, we introduce an extension to the original Voronoi diagram by extending the boundary of each Voronoi region by the *extension radius* r associated with each point p where


p
VR(p)
Perpendicular bisector
p
d1
A1
S
d2
A2
A3
Probabilistic nearest alarm region
A1
A2
r
r
A3
r
r
Deterministic nearest alarm region
A4

(a) Voronoi Diagram (b) Voronoi Extension (c) Voronoi Extension

Fig. 3. Nearest Alarms-based Grouping

$r = \sqrt{(\frac{l}{2})^2 + (\frac{h}{2})^2}$. l, h denote the length and height of the alarm region associated with center point p. The extended Voronoi regions for alarms A_1, A_2, A_3 and A_4 are shown in Figure 3(c). Extending the Voronoi region boundaries leads to overlaps among neighboring Voronoi regions; subscribers inside overlapping regions (*probabilistic nearest alarm region*) may have more than one possible nearest alarm whereas subscribers inside non-overlapping regions (*deterministic nearest alarm region*) can have only one nearest alarm.

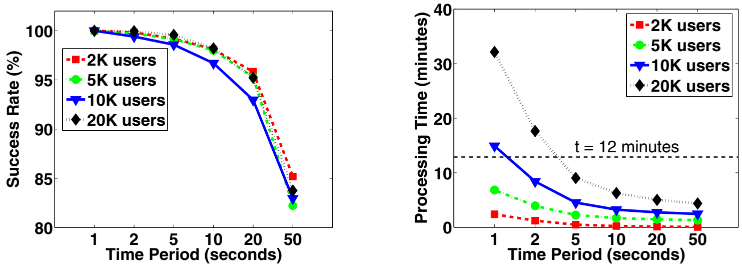
Nearest alarm grouping is efficient for systems that have infrequent addition or removal of alarms and have no hotspots. However, it fails when there is frequent addition/removal of spatial alarms, since Voronoi diagrams need to be reconstructed each time an alarm is added or removed. In addition, high density of alarms in some areas may also lead to large overlaps among Voronoi regions, reducing the efficiency of this technique.

4 Experimental Evaluation

In this section, we report our experimental evaluation results. We show that our safe period-based framework and optimization techniques for spatial alarm processing are scalable while maintaining high accuracy.

4.1 Experimental Setup

Our simulator generates a trace of vehicles moving on a real-world road network using maps available from the National Mapping Division of the U.S. Geological Survey (USGS [4]) in Spatial Data Transfer Format (SDTS [3]). Vehicles are randomly placed on the road network according to traffic densities determined from the traffic volume data in [9]. We use a map from Atlanta and surrounding regions of Georgia, which covers an area larger than 1000 km^2, to generate the trace. Our experiments use traces generated by simulating vehicle movement for a period of fifteen minutes, results are averaged over a number of such traces. Default traffic volume values allow us to simulate the movement of a set of 20,000 vehicles. The default spatial alarm information consists of a set of 10,000 alarms installed uniformly over the entire map region; around 65% of the alarms are private, 33% shared and the rest are public alarms.



(a) Success Rate

(b) Alarm Processing Time

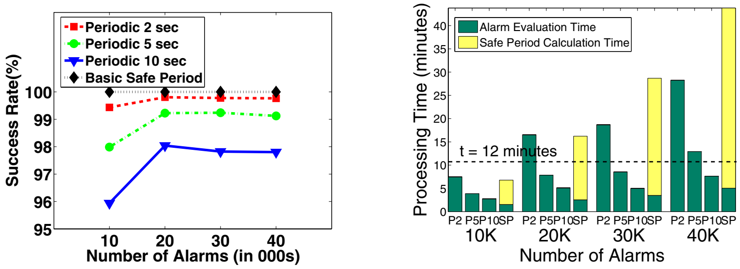
Fig. 4. Scalability with Varying Number of Users

4.2 Experimental Results

The first set of experiments measures the performance of periodic alarm evaluation by varying the time period of updates and shows that this approach does not scale. The second set of experiments compares the basic safe period approach against periodic evaluation and shows that safe period-based alarm processing offers higher success rate with lower evaluation time. The last set of experiments compares the performance of the various grouping-based optimizations against the basic safe period approach exhibiting the scalability of our grouping optimizations.

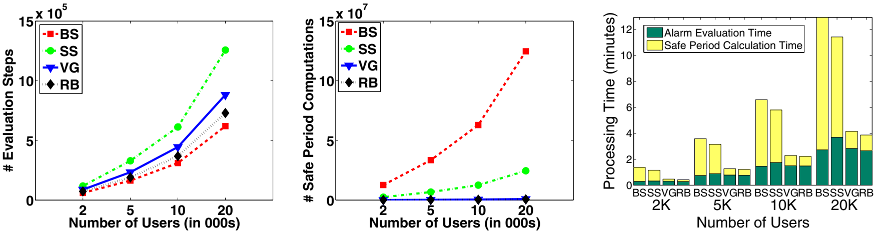
Scalability Problems of Periodic Alarm Evaluation Technique: In this first set of experiments, we measure the scalability of the periodic alarm evaluation technique with varying number of users. Figure 4 displays the results as we vary the number of users from 2K to 20K. The time period t_p for periodic alarm evaluation is varied from 1 second to 50 seconds. As can be seen from Figure 4(a), the success rate for alarm evaluation is 100% only if t_p= 1 second; for higher t_p success rate starts falling, even with t_p= 2 seconds the success rate does fall to 99.9% which may not be acceptable from QoS viewpoint as this translates to a significant number of alarm misses. The sequence of alarms to be triggered for 100% success rate are determined from a trace generated with highly frequent location updates for each user. The alarm processing time is plotted in Figure 4(b). Our traces are of fifteen minutes duration; considering that the system has around 80% of this time for processing spatial alarms we set the maximum processing time available to the system at t=12 minutes as indicated by the horizontal dotted line in Figure 4(b). For 10K users the system is unable to process alarms at t_p=1 seconds, thus failing to attain 100% success rate. For 20K users, this scalability problem becomes worse and the system is able to evaluate alarms only at t_p=5 seconds. Thus, we conclude that periodic evaluation approach does not scale.

Performance Comparison with Basic Safe Period Approach: In this section, we compare the performance of basic safe period approach against periodic evaluation. We display the results for periodic approach with t_p=2 seconds,



(a) Success Rate (b) Alarm Processing Time

Fig. 5. Safe Period Optimization with Varying Number of Alarms



(a) Alarm Evaluation Steps (b) Safe Period Computations (c) Alarm Processing Time

Fig. 6. Safe Period Optimizations with Varying Number of Users

t_p= 5 seconds, t_p=10 seconds and the basic safe period optimization as discussed in Section 2 (P2, P5, P10 and SP in Figure 5(b)). Figure 5 displays the success rate and processing time as we increase the number of alarms from 10K to 40K. Figure 5(a) displays that the success rate is 100% for basic safe period approach and all periodic approaches miss at least a few alarm triggers. Figure 5(b) displays the alarm processing time for P2, P5, P10 and SP with varying number of alarms. The alarm processing time, as shown in Figure 5(b), displays the inability of our basic safe period approach to scale to large number of alarms. In presence of even 20K installed alarms, the approach has excessive safe period computation time which pushes the overall processing time beyond the 12 minute limit determined earlier. Our alarm grouping and subscriber mobility-based techniques provide optimizations to overcome this problem.

Scalability of Safe Period Evaluation Techniques: We now discuss the performance of the safe period optimization techniques to test the scalability of our framework. Figure 6 shows the number of alarm evaluation steps, number of safe period computations and the alarm processing time required by each approach: Basic Safe Period Optimization (BS), Subscriber-Specific Spatial Locality (SS), Voronoi Grid-Based (VG) and a Range-based Subscriber-Specific Grouping Optimization (RB). VG and RB approaches consider alarms only in the vicinity of the current subscriber position for safe period computation. Results for Spatial Locality-based grouping show expected trends but this approach has high overall processing time as the system needs to perform significant amount of computation to determine relevance of alarms/alarm groups for each subscriber. Hence, we exclude this approach from the results.

Figure 6(a) displays the number of alarm evaluation steps required by each approach. Basic safe period measures the safe period to each relevant alarm and uses this safe period to avoid further evaluations. As a result, this approach has to perform a low number of alarm evaluations but each evaluation step involves a very large number of safe period computations. Hence the number of safe period computations for this approach is extremely large (Figure 6(b)) which makes this approach overall computationally expensive as can be seen from the total alarm processing times in Figure 6(c). Subscriber-specific spatial locality grouping incurs a large number of alarm evaluation steps as can be seen from Figure 6(a). This approach first evaluates safe period for each alarm group; once the user enters an alarm monitoring region another evaluation step is required to determine the safe period to all alarms lying within the alarm monitoring region. Further, this approach needs to keep a check on subscriber position inside the alarm monitoring region and switch to per alarm group-based safe period computations once the subscriber moves outside the current alarm monitoring region. These additional evaluation steps imply that this approach will incur a larger number of alarm evaluation steps with each evaluation step requiring a small number of safe period computations: either for each alarm group or for all alarms lying within the current alarm monitoring region. Thus the number of safe period computations required by this approach is much lower than the basic approach despite the larger number of alarm evaluation steps. Consequently, the overall processing time for SS is lower than the BS approach as can be seen from Figure 6(c). The VG and RB approaches lower the number of alarm evaluation steps by considering only alarms or alarm groups in the vicinity of the client. In this set of experiments, the RB approach considers alarms within a radius of 1000m from the client position. VG approach overlays a grid with cell size 1000m × 1000m on top of the Voronoi extension and considers alarms only within the current subscriber grid cell. The number of evaluation steps for these approaches is still larger than the number of evaluation steps used by the basic approach as the safe periods computed by this approach may be lower than the safe period computed by the basic approach, in case no relevant alarms/alarm groups lie within the radius range or the current grid cell of the subscriber. However, each alarm evaluation step involves a very small number of safe period calculations leading to an extremely small number of safe period computations (in Figure 6(b) results for VG and RB are overlapping and values are much smaller than other two approaches). Consequently, the overall processing times for these two approaches are significantly lower than other approaches. From these results we can conclude that our safe period optimizations significantly aid the scalability of the system.

5 Related Work

An event-based location reminder system has been advocated by many human computer interaction projects [8,10,12,13,14]. Understandably, the primary focus of the work is from the point of view of the usability of such systems. None of these approaches deal with the system oriented issues which need to be resolved to make such systems feasible. In the realm of information monitoring, event-based systems have been developed to deliver relevant information to users on demand [7,11]. In addition to monitoring continuously evolving user information needs, spatial alarm processing systems also have to deal with the complexity of monitoring user location data in order to trigger relevant alerts in a non-intrusive manner. Applications like Geominder [1] and Naggie [2] already exist which provide useful location reminder services using cell tower ID and GPS technology, respectively. Client-based solutions for spatial alarm processing should focus on efficiently evaluating spatial alarms while preserving client energy. Our server-centric architecture makes it possible for users to share alarms and make use of external location information monitoring services which provide relevant location-based alerts. A server-centric approach is also essential for extending the technology to clients using cheap location detection devices which may not possess significant computational power.

6 Conclusion

The paper makes two important contributions towards supporting spatial alarm-based mobile applications. First, we introduce the concept of *safe period* to minimize the number of unnecessary alarm evaluations, increasing the throughput and scalability of the system. Second, we develop a suite of spatial alarm grouping techniques based on spatial locality of the alarms and motion behavior of the mobile users, which reduces the safe period computation cost for spatial alarm evaluation at the server side. We evaluate the scalability and accuracy of our approach using a road network simulator and show that the proposed safe period-based approach to spatial alarm processing offers significant performance enhancements for alarm processing on server side while maintaining high accuracy of spatial alarms.

Acknowledgements

This work is partially supported by grants from NSF CyberTrust, NSF SGER, NSF CSR, AFOSR, IBM SUR grant and IBM faculty award. The authors would like to thank Bugra Gedik for providing the mobile object simulator.

References

1. Geominder, http://ludimate.com/products/geominder/
2. Naggie 2.0: Revolutionize Reminders with Location, http://www.naggie.com/
3. Spatial Data Transfer Format, http://www.mcmcweb.er.usgs.gov/sdts/

4. U.S. Geological Survey, http://www.usgs.gov
5. Aurenhammer, F.: Voronoi Diagrams–A Survey of a Fundamental Geometric Data Structure. ACM Computing Surveys 23(3), 345–405 (1991)
6. Bamba, B., Liu, L., Yu, P.S.: Scalable Processing of Spatial Alarms. Technical Report, Georgia Institute of Technology (2008)
7. Bazinette, V., Cohen, N., Ebling, M., Hunt, G., Lei, H., Purakayastha, A., Stewart, G., Wong, L., Yeh, D.: An Intelligent Notification System. IBM Research Report RC 22089 (99042) (2001)
8. Dey, A., Abowd, G.: CybreMinder: A Context-Aware System for Supporting Reminders. In: Second International Symposium on Handheld and Ubiquitous Computing, pp. 172–186 (2000)
9. Gruteser, M., Grunwald, D.: Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking. In: MobiSys (2003)
10. Kim, S., Kim, M., Park, S., Jin, Y., Choi, W.: Gate Reminder: A Design Case of a Smart Reminder. In: Conference on Designing Interactive Systems, pp. 81–90 (2004)
11. Liu, L., Pu, C., Tang, W.: WebCQ - Detecting and Delivering Information Changes on the Web. In: CIKM, pp. 512–519 (2000)
12. Ludford, P., Frankowski, D., Reily, K., Wilms, K., Terveen, L.: Because I Carry My Cell Phone Anyway: Functional Location-Based Reminder Applications. In: SIGCHI Conference on Human Factors in Computing Systems, pp. 889–898 (2006)
13. Marmasse, N., Schmandt, C.: Location-Aware Information Delivery with ComMotion. In: Thomas, P., Gellersen, H.-W. (eds.) HUC 2000. LNCS, vol. 1927, pp. 157–171. Springer, Heidelberg (2000)
14. Sohn, T., Li, K., Lee, G., Smith, I., Scott, J., Griswold, W.: Place-Its: A Study of Location-Based Reminders on Mobile Phones. In: Beigl, M., Intille, S.S., Rekimoto, J., Tokuda, H. (eds.) UbiComp 2005. LNCS, vol. 3660. Springer, Heidelberg (2005)

Coverage Based Expanding Ring Search for Dense Wireless Sensor Networks

Kiran Rachuri, A. Antony Franklin, and C. Siva Ram Murthy*

Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai-600036, India
{kiranr,antony}@cse.iitm.ac.in, murthy@iitm.ac.in

Abstract. Expanding Ring Search (ERS) is a prominent technique used for information discovery in multi-hop networks where the initiator of search is unaware of any of the γ locations of the target information. ERS reduces the overhead of search by successively searching larger number of hops starting from the location of search initiator. Even though ERS reduces overhead of search compared to flooding, it still incurs huge cost which makes it unsuitable especially to energy constrained networks like Wireless Sensor Networks (WSNs). Moreover, the cost of search using ERS increases with node density, which limits its scalability in densely deployed WSNs. In this paper, we apply the principles of area coverage to ERS and propose a new protocol called Coverage Based Expanding Ring Search (CERS(k), where k is the amount of parallelism in search) for energy efficient and scalable search in WSNs. CERS(k) is configurable in terms of energy-latency trade-off which enables it applicable to varied application scenarios. The basic principle of CERS(k) is to route the search packet along a set of ring based trajectories that minimizes the number of messages transmitted to find the target information. We believe that query resolution based on the principles of area coverage provides a new dimension for conquering the scale of WSN. We compare CERS(k) with the existing query resolution techniques for unknown target location such as, ERS, Random walk search, and Gossip search.

1 Introduction

Wireless Sensor Networks (WSNs) consist of a huge number of tiny sensor nodes and one or more sink nodes. Sensor nodes are battery powered and possess very limited computation and communication capabilities. Therefore the energy of sensor nodes should be used judiciously. Sink nodes are powered and are storage points for most of the data emerging from environmental sensing of sensor nodes. The authors of [1], based on how data is gathered, categorize WSNs as follows:

- PUSH or CONTINUOUS COLLECTION: Sensor nodes periodically sense environment and send data to the sink node.

* Author for correspondence.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 245–256, 2008.

- PULL or QUERYING: Sensor nodes sense environment and store the information locally. On need basis, the sink node queries for the required information.
- PUSH-PULL: This paradigm involves both PUSH and PULL. Sensor nodes push the sensed events to different sensor nodes in the terrain in a predetermined way that is used by the search initiator for finding the target information.

Usage of PUSH, PULL, or PUSH-PULL approach depends on various factors such as, application requirements, memory constraints, and energy savings. As sensor nodes are battery powered, energy is a premium resource and in most cases energy efficiency is always a requirement irrespective of the other requirements. PUSH approach is efficient when continuous sensing is required and PULL approach is efficient for low frequency data gathering. In PULL paradigm, WSN can be considered as a distributed database and on need basis, the sink node sends queries for data collection. Some of the factors that influence the usage of PUSH-PULL approaches are the rate of occurrence of events, the query rate, the type of events sensed, and available memory resources on sensor nodes. If the query rate is low and the rate of occurrence of events is high or event type is audio or video then it is clearly not feasible to store them in multiple sensor nodes as they may consume the memory completely.

Querying or searching in WSNs is an active research area and there are many proposals for reducing the overhead of search cost [1,2,3] based on the query type. The authors of [4] classify the types of WSN queries as the following:

- Continuous vs one-shot queries.
- Aggregate vs non-aggregate queries.
- Complex vs simple queries.
- Queries for replicated data vs queries for unique data.

There are various proposals for the above listed query types such as, Directed Diffusion routing [5] for continuous and aggregate queries, Acquire mechanism [4] for one-shot and complex queries for replicated data, and Random walk [6] for simple one-shot queries.

In this paper, we focus on PULL and UNSTRUCTURED [7] WSNs where, the sink node sends simple and continuous/one-shot queries for replicated data. In UNSTRUCTURED WSNs, the search initiator has no clue about the location of target information. In this paper, the meaning of replicated data is different from the one used in PUSH-PULL WSNs. In PUSH-PULL, events are replicated and stored in multiple sensor nodes but, we consider a case where events occur uniformly in the terrain and events of a type occur at γ locations [8] which are uniformly distributed in the given terrain. In PUSH-PULL, an event is replicated in a pre-determined way at γ locations (by the sensor node which sensed it) and in our case, an event of a type occurs at γ locations. An example of simple and continuous query for replicated data is to find a sensor node which sensed an event of a type and then request a stream of data to be sent from that sensor node to the sink node. This involves searching for at least one sensor node which sensed an event of the target event type, and that event type has occurred at multiple places in the terrain and is stored locally by multiple sensor nodes. Once a target node is found, a continuous stream of data is sent from that sensor node to the sink node for a certain amount of time. Since this scenario involves sending continuous data, the replica should be the nearest one to the search initiator, otherwise the energy consumed for transmitting the information continuously from the sensor node to the sink node will be huge. We refer to the information to be searched as the target information and a node which has the target information as a target node. Based on the above example, we now define two types of cost *viz.*, *SearchCost*, which is the cost of search and *DataTransferCost*, which is the cost of transmitting data from the discovered target node to the sink node. For a fixed amount of data to be transferred, the *DataTransferCost* is directly proportional to the proximity of the discovered target node to the sink node. Therefore, the search techniques should not only reduce the *SearchCost* but also find the nearest replica of the target information to minimize the *DataTransferCost*.

One of the prominent techniques used for searching in PULL and UNSTRUCTURED WSNs is Expanding Ring Search (ERS) [2,3]. ERS reduces the overhead of search by successively searching larger number of hops starting from the location of search initiator and finds the nearest replica of the target information. Even though ERS reduces overhead of search compared to flooding, it still incurs huge cost which makes it unsuitable especially to energy constrained networks like WSNs. Moreover, the cost of search using ERS increases with node density, which limits its scalability in densely deployed WSNs. In this paper, we apply the principles of area coverage to ERS and propose a new protocol called Coverage Based Expanding Ring Search (CERS(k)) for energy efficient and scalable search in WSNs. CERS(k) not only consumes less cost than ERS but also is independent of node density which makes it scalable especially in densely deployed WSNs.

CERS(k) operates by dividing the terrain into concentric *rings*. The width of each *ring* is twice the transmission radius of sensor nodes. Each *ring* is characterized by a *Median* which is the circular trajectory that divides the *ring* into two halves of equal width (*i.e.*,width = transmission radius of sensor nodes). Based on the value of parameter k, CERS(k) searches the *rings* either sequentially or in parallel for finding the target information. CERS(k) searches for the target information until it is found or all *rings* are searched. In each *ring*, by exploiting the localization capabilities of sensor nodes, query packet travels along the *Median* of *ring* via beacon-less forwarding to cover the entire area of *ring*. When a target node receives the query packet, a response packet is sent back to the sink node. For a fixed terrain, the number of transmissions required to cover the entire terrain area is constant due to which, the cost of CERS(k) is independent of node density for a given terrain size. Hence, CERS(k) is highly advantageous for densely deployed WSNs. We prove that the cost of CERS(k) is independent of node density via simulations in Section 5.

2 Assumptions

- The terrain is considered to be circular. The sink node is static and is placed at the center of the circular terrain.
- Sensor nodes are stationary and are deployed uniformly in the terrain.
- Sensor nodes are aware of their own location coordinates. Since sensor nodes are stationary, assigning location coordinates to sensor nodes is a one time task and is part of the initial setup of WSN.
- We consider PULL and UNSTRUCTURED WSN where the sink node (search initiator) sends simple, one-shot/continuous queries for replicated data.
- The search initiator is unaware of the locations of target information replicas.
- Events occur uniformly in the given terrain.
- To relay the search packet along the *Medians* of the *rings*, we assume that the density of sensor nodes is high.

3 Related Work

In [1], the authors propose a PUSH-PULL strategy for information dissemination and gathering where the storage is replicated at more than one node. The topology is considered to be static square grid with the sink node located at the bottom corner. In Trajectory Based Forwarding (TBF) [9], the authors propose a general framework for routing packets via a predefined curve or a trajectory. TBF is related to our work as CERS(k) also involves routing search packets over *rings*. But the authors of TBF have not presented analysis or simulation results for any specific trajectory, to fully understand its benefits in terms of energy efficiency and scalability. In [8], the authors propose a biased walk which resembles a spiral walk for information discovery in WSNs. The query packet visits all nodes whose distance from the sink node is less than the distance between the discovered target node and the sink node. Due to this, the protocol does not scale well in dense WSNs.

For UNSTRUCTURED WSNs where the sink node is not aware of the location of target information, search proceeds blindly for tracing the target information. The following are the most widely used techniques for searching in UNSTRUCTURED WSNs: ERS [2,3], Random walk search [6], and Gossip search [10]. ERS avoids network-wide broadcast by searching for the target information with increasing order of TTL (Time-To-Live) values. TTL limits the number of hops to be searched from the source node. If search fails continuously up to TTL *Threshold* hops, ERS initiates network-wide broadcast. The main disadvantage of this protocol is that it resembles flooding in the worst case.

In Random walk search, when a node has to forward the search packet, it randomly selects one of its neighbors and forwards the search packet to the selected neighbor. The basic idea here is the random wandering in the network in search of the target information until TTL (number of hops) is expired or the target information is found. The main disadvantage of Random walk is that the probability of finding the nearest replica of the target information is low and due to this, the *DataTransferCost* will be very high especially in the case of continuous queries.

In Gossip search, the source node broadcasts the search packet and all receivers either forward it with a probability p or drop it with a probability $1 - p$. In some cases, gossip dies early without reaching reasonable number of nodes even for higher values of p which increases the non-determinism of Gossip search. For this reason, in [10], the authors propose $GOSSIP(p, k)$ where k is the number of hops for which the search packet has to be transmitted with probability 1 *i.e.*, for first k hops the search packet is always forwarded after which it is forwarded with a probability p. The main disadvantage of Gossip search is that of sending message to most of the sensor nodes even when the target information is located close to the source node.

4 Protocol Design

The basic principle of CERS(k) is that if a subset of the total sensor nodes transmit the search packet such that, the entire circular terrain area is covered by these transmissions, then at least one target node will definitely receive the search packet. However, if the search packet is broadcasted to the entire circular terrain, even though we find the target information, the number of messages transmitted will be huge. To minimize the number of message transmissions, CERS(k) divides the circular terrain into concentric *rings* such that if the area in all these *rings* are covered, then the entire area of circular terrain will be covered. In CERS(k), the concentric *rings* are searched either sequentially or in parallel depending on the value of k until the target information is found or all the *rings* are searched. Each *ring* is of width equal to twice the transmission range of sensor nodes. The *Median* of a *ring* is the circular trajectory that divides the *ring* into two concentric rings of width equal to the transmission radius (r) of sensor nodes. In Figure 1, the *Medians* of *rings* are shown as solid lines and the borders of *rings* are shown as dotted lines.

The fields of the search packet used by CERS(k) are listed in Table 1 and the terminology used in explaining CERS(k) is listed in Table 2. When CERS(k) is

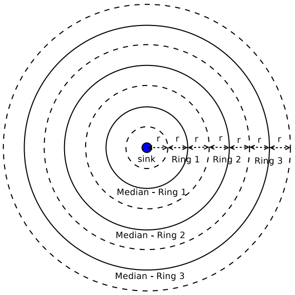


Fig. 1. Circular terrain divided into concentric rings

Table 1. Search Packet Fields of CERS(k)

(rX, rY)	Location coordinates of *RelayedNode*.
(dX, dY)	Location coordinates of *DestinationPoint*.
SeqNo.	Sequence number of search packet, required to avoid duplicate forwards.
Radius	Radius of the current *ring*.
Target	Target event information.
Angle	Angle constraints for relaying.

Table 2. CERS(k) Terminology

SinkNode	Node which initiated the search.
RelayedNode (R)	Node which relayed the search packet after receiving from previous *Relayed/SinkNode*.
CurrentNode (C)	Node which received the search packet from *RelayedNode*.
DestinationPoint (D)	Next destination point on the current *ring*.
$Dist_{r,d}$	Distance between *RelayedNode* and *DestinationPoint*.
$Dist_{c,d}$	Distance between *CurrentNode* and *DestinationPoint*.
$Dist_{c,m}$	Distance between *CurrentNode* and the *Median* of *ring*.

initiated for searching the target information, the sink node broadcasts search packet by embedding the identification (*Radius*) of first *ring* in it, with $Angle = 30°$. This search packet travels along the *Median* of the first *ring*, starting at a point on it and ending at a point close to the starting point. The *Median* of first *ring* is represented by the circle equation with the center as sink node and radius equal to twice the transmission range of sensor nodes. In a *ring* the search packet is advanced in increments of transmission radius of sensor nodes based on the following logic: A node which relays the search packet is called *RelayedNode*. Each *RelayedNode* embeds the information about the *DestinationPoint* in the search packet. The *DestinationPoint* is a point on the *Median* of *ring* at a distance of r from the *RelayedNode*. A node which receives the search packet from *RelayedNode* is referred as *CurrentNode*. All *CurrentNodes* evaluate the following two conditions to check whether they are eligible to forward the search packet or not:

1. $Dist_{c,d} < Dist_{r,d}$
2. $\angle DRC < Angle$

A node which satisfies both these conditions is referred as *EligibleNode*. The first condition makes sure that the *EligibleNode* is closer to *DestinationPoint* compared to *RelayedNode*. The second condition makes sure that if one of the *EligibleNodes* relays the search packet, all other *EligibleNodes* can receive this packet and drop the packet which should be relayed by them. An *EligibleNode* has to wait for a time proportional to its proximity to the *DestinationPoint* and the *Median* of *ring* before relaying the packet. The time to wait before relaying is given by

$$T_{wait} \propto Dist_{c,d} + Dist_{c,m}$$

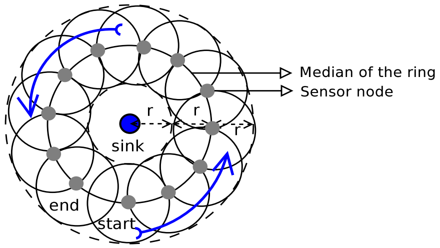


Fig. 2. Area covered by a single ring (1^{st} ring)

The *EligibleNode* which is closer to the *Median* of current *ring* and closer to the *DestinationPoint* will have lesser waiting time compared to other *EligibleNodes*. The *EligibleNode* with smaller waiting time relays the search packet while others drop it. This continues until the search packet travels the entire length of the *Median* of *ring* or there is no other node to relay the search packet further. The idea behind forwarding via the *Median* of *ring* is to cover the entire region of *ring* as illustrated in Figure 2. The protocol used for forwarding the search packet is similar to the position based beacon-less routing [11] with some modifications. If there are no nodes eligible to relay the search packet, the *RelayedNode* re-broadcasts the search packet without angle constraints *i.e.*, $Angle = 360°$. This might result in an increase in the number of *EligibleNodes* and it is possible that when an *EligibleNode* relays the search packet, all other *EligibleNodes* may not receive the search packet. As a result of this, there might be multiple *EligibleNodes* relaying the search packet. This is a compromise as there are no *EligibleNodes* to relay the search packet. Under high node density, the probability of not finding *EligibleNodes* with 30° angle constraint is very less and we validate this via simulations in Section 5. When a target node is in the region of current *ring*, it receives the search packet and responds to the sink node by sending a response packet. The sink node continues to search until all the *rings* are searched or the target information is found. From Figure 2, one can observe that some portions of the *ring* are not covered by transmissions of any of the sensor nodes. Because of this, CERS(k) is not a deterministic protocol but, the probability of finding target information is very high under high node density, which will be shown via simulations in Section 5.

In CERS(k), the parameter k determines the parallelism in searching *rings*. In particular, k determines the number of *rings* to be searched in parallel. CERS(1) searches the *rings* one after other, due to which the search cost is low but latency is high. CERS(2) searches in two *rings* at a time until the target information is discovered. As the k value increases, the cost of search using CERS(k) increases and latency decreases. Let N be the total number of *rings* for a given terrain, then CERS(N) searches with utmost parallelism *i.e.*, searches all *rings* in one iteration. The case of CERS(N) is best in terms of latency of search and worst in terms of cost of search. Since the value of k is instrumental in cost-latency trade-off, its value should be carefully chosen based on the application requirements. We will validate these claims via simulations in the next section.

5 Simulations

To validate our claims in Section 4, we simulate CERS(k), ERS, Random walk, and Gossip protocols in the ns-2 [12], a popular discrete event network simulator.

5.1 Performance Metrics Used

- *Number of transmitted bytes (Cost of search)*: Average number of bytes transmitted for finding the target information. As the message formats are not uniform across protocols, we measured the number of bytes transmitted instead of the number of messages transmitted.
- *Energy consumed*: The total energy consumed for finding the target information. This includes the number of bytes transmitted and received for finding the target information.
- *Latency*: Time taken to find the target information.
- *Probability of finding the target information*: Probability of finding the target information is a measure of non-determinism for search protocols.
- *Number of hops between the sink node and the discovered target node*: This metric reflects the proximity of the discovered target node and the sink node. *DataTransferCost* is directly proportional to this metric.

5.2 Simulation Setup

We consider a circular terrain where sensor nodes are uniformly deployed and the sink node is placed at the center of terrain. The transmission radius of sensor nodes is fixed at $30m$. The *TTL Threshold* value for ERS is fixed at 3 as this value is found to be optimum [2]. The Gossip probability is fixed at 0.65 as this value is found to gossip the search message to most of the nodes in WSN [10]. All the graphs for the performance metrics are plotted for 95% confidence level. Simulations are performed for the following scenarios: (i) **Node density variation**: We consider a WSN with fixed terrain radius ($200m$) and fixed number of target replicas (5) and, the number of nodes is varied from 200 to 1200 in increments of 200. (ii) **Terrain size variation**: We consider a WSN with fixed node density (0.00884 $nodes/sq.m.$) and fixed number of target replicas (5) and vary the terrain radius from $75m$ to $200m$ in increments of 25.

5.3 Simulation Results

Node density variation: Figure 3 shows the effect of increase in node density on the number of bytes transmitted to find the target information. We can observe that the number of bytes transmitted is lowest for CERS(k). For ERS and Gossip protocols, it increases linearly with node density whereas for Random walk protocol, it is invariant to node density. Based on this result, we can infer that the cost of search using CERS(k) and Random walk are invariant to node density and between the two, CERS(k) incurs less cost. Even though Random walk fares well with respect to this metric, it finds inferior paths to target replicas

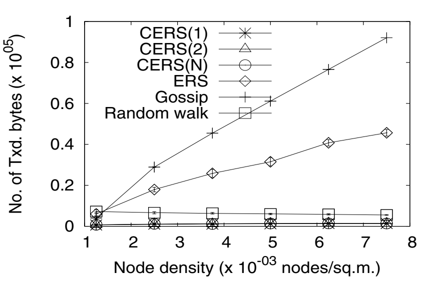


Fig. 3. Node density vs no. of Txd. bytes

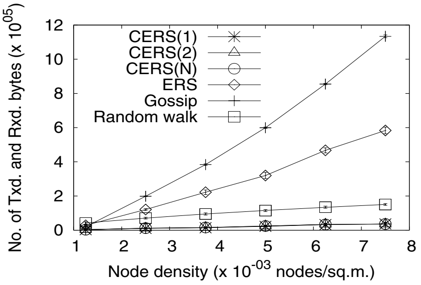


Fig. 4. Node density vs no. of Txd. and Rxd. bytes

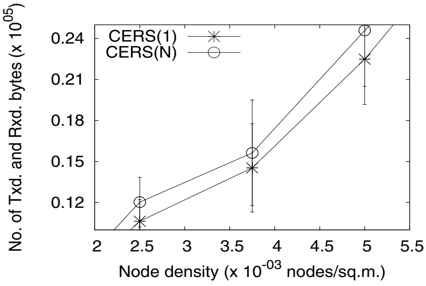


Fig. 5. Node density vs no. of Txd. and Rxd. bytes for CERS(k)

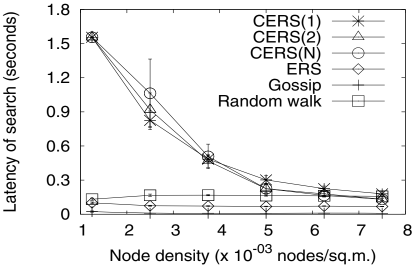


Fig. 6. Node density vs latency of search

which increases the *DataTransferCost*. This fact will be presented later in this section.

Figure 4 shows the effect of increase in node density on the total number of transmitted and received bytes. As the node density increases, the energy consumption of CERS(k) is much lower compared to the other protocols. The increasing trend in energy consumption for all protocols is due to increase in number of nodes receiving the search packet with increase in node density. Figure 5 shows the effect of k on the total number of transmitted and received bytes using CERS(k) (this figure is zoomed version of Figure 4, due to which the confidence intervals are so high). We can observe that the energy consumption of search using CERS(k) increases with k.

Figure 6 shows the latency of the search techniques. It is clear from the results that the difference in latencies of the search techniques is less at high node density values. At low node density values, CERS(k) takes more time than the existing search techniques and this can be attributed to failure in finding the *EligibleNodes*. As node density increases, the latency of CERS(k) comes very close to the existing protocols. Gossiping is the fastest of all as there is only one iteration in which the search packet is broadcasted to most of the nodes in the network. In ERS, there is wait time involved for each ring until $TTL\ Threshold$ and this slows down the protocol compared to Gossip. From Figure 7, we can observe that the latency of CERS(k) decreases with increase in k value (this

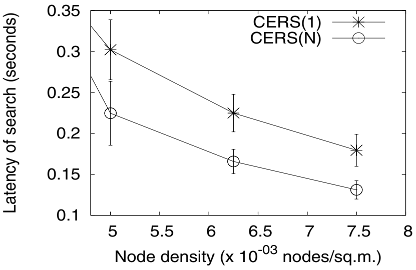


Fig. 7. Node density vs latency of search for CERS(k) protocol

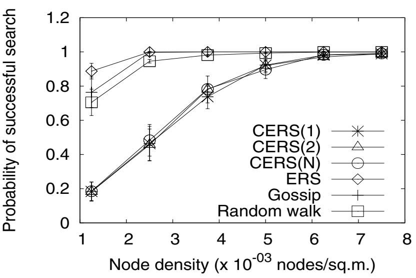


Fig. 8. Node density vs probability of successful search

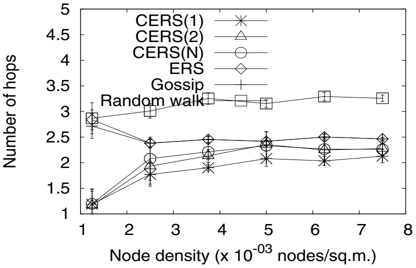


Fig. 9. Node density vs proximity of discovered target to sink node

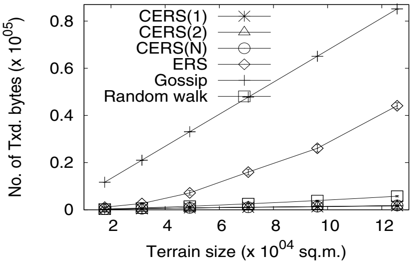


Fig. 10. Terrain size vs no. of Txd. bytes

figure is zoomed version of Figure 6) which is in accordance with our claims in Section 4.

Figure 8 shows the probability of successfully finding the target information. When the node density is low, it is quite possible that none of the target nodes are connected because of which the probability of successful search is very less. As the node density increases, the probability increases and reaches unity for high node density values. The probability of finding the target information using CERS(k), reaches unity at a slower rate compared to ERS, Gossip, and Random walk protocols. This is due to failure in finding *EligibleNodes* to relay the search packet at lower densities. This shows that the non-determinism of CERS(k) is very less and it almost finds the target information at high node densities.

Figure 9 shows that the number of hops from the sink node to the discovered target node is between 2 and 2.5 for CERS(k), ERS, and Gossip protocols whereas for Random walk, it is higher. This shows that even though the cost of Random walk is invariant to node density, the target replicas discovered are inferior in terms of proximity to the sink node because of which, the *DataTransferCost* will be very high for continuous type of queries. We can also observe that the values of *Number of hops* for ERS and Gossip protocols are slightly higher than CERS(k). This can be attributed to failure in finding the closest replica of target information on some occasions because of collisions. However,

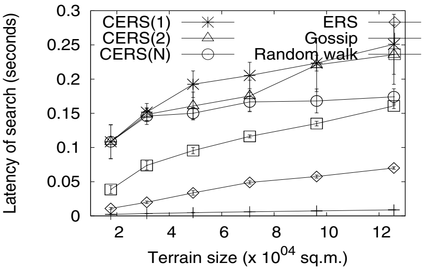


Fig. 11. Terrain size vs latency of search

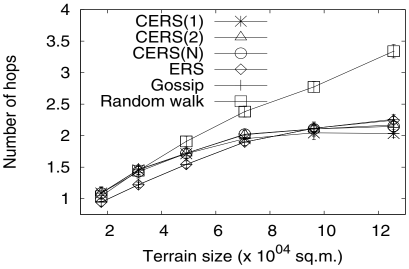


Fig. 12. Terrain size vs proximity of discovered target to sink node

the probability of ERS and Gossip not finding the closest replica of the target information is very less.

Terrain size variation: From Figure 10, we can infer that the cost of all the search techniques increases with increase in terrain size. However, CERS(k) consumes the least energy of all the search techniques, irrespective of the terrain size. Figure 11 shows that the latency of CERS(k) is very close to the existing proposals *i.e.*, the average latency of CERS(k) is in the order of $200ms$. The latency of CERS(k) decreases with increase in k value as the parallelism of search increases. Figure 12 shows that the number of hops between the discovered target node and the sink node for CERS(k), ERS, and Gossip protocols is almost at the same level and is lower than Random walk. Moreover, for Random walk, it increases linearly with terrain size. This is a big disadvantage of Random walk protocol for continuous type of queries.

6 Conclusion

In this paper, we presented CERS(k), an energy efficient and scalable query resolution protocol for simple, one-shot/continuous queries for replicated data. The conclusion drawn from the paper is that under high node density, CERS(k) consumes less energy compared to the existing search techniques such as, ERS, Random walk, and Gossip protocols and it is unaffected by the variation in node density. Random walk comes very close to CERS(k) in terms of cost of search but, due to the inferior target replicas (in terms of proximity to the sink node) discovered by the Random walk, the *DataTransferCost* will be huge and makes it unsuitable for resolving continuous type of queries. The parameter k in CERS(k) can be adjusted to balance cost and latency and this makes it applicable to vast and varied application scenarios. We believe that query resolution based on the principles of area coverage provides a new dimension for enhancing the scalability of query protocols in WSN. The following are the main advantages of CERS(k):

- Energy is the most premium resource in WSNs and CERS(k) achieves significant energy savings for dense WSNs.

- Cost of search is unaffected by variation in node density, which makes CERS(k) scalable for highly dense WSNs.
- The cost-latency trade-off can be adjusted using k based on the application requirements.

References

1. Liu, X., Huang, Q., Zhang, Y.: Combs, needles, haystacks: Balancing push and pull for discovery in large-scale sensor networks. In: SenSys 2004: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, November 2004, pp. 122–133 (2004)
2. Hassan, J., Jha, S.: Optimising expanding ring search for multi-hop wireless networks. In: GLOBECOM 2004: Proceedings of the 47th IEEE Global Telecommunications Conference, November 2004, pp. 1061–1065 (2004)
3. Chang, N.B., Liu, M.: Controlled flooding search in a large network. IEEE/ACM Transactions on Networking 15(2), 436–449 (2007)
4. Sadagopan, N., Krishnamachari, B., Helmy, A.: The ACQUIRE mechanism for efficient querying in sensor networks. In: SNPA 2003: Proceedings of the 1st IEEE International Workshop on Sensor Network Protocols and Applications, May 2003, pp. 149–155 (2003)
5. Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., Silva, F.: Directed diffusion for wireless sensor networking. IEEE/ACM Transactions on Networking 11(1), 2–16 (2003)
6. Tian, H., Shen, H., Matsuzawa, T.: Random walk routing for wireless sensor networks. In: PDCAT 2005: Proceedings of the 6th International Conference on Parallel and Distributed Computing Applications and Technologies, December 2005, pp. 196–200 (2005)
7. Ahn, J., Krishnamachari, B.: Modeling search costs in wireless sensor networks. Tech. Rep. CENG-2007-1, Computer Science Department, University of Southern California (2007)
8. Huang, H., Hartman, J., Hurst, T.: Data-centric routing in sensor networks using biased walk. In: SECON 2006: Proceedings of the 3rd IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, September 2006, pp. 1–9 (2006)
9. Niculescu, D., Nath, B.: Trajectory based forwarding and its applications. In: MobiCom 2003: Proceedings of the 9th International Conference on Mobile Computing and Networking, September 2003, pp. 260–272 (2003)
10. Haas, Z.J., Halpern, J.Y., Li, L.: Gossip-based ad hoc routing. IEEE/ACM Transactions on Networking 14(3), 479–491 (2006)
11. Uttel, M., Braun, T.: A novel position-based and beacon-less routing algorithm for mobile ad-hoc networks. In: ASWN 2003: Proceedings of the 3rd IEEE Workshop on Applications and Services in Wireless Networks, July 2003, pp. 197–210 (2003)
12. The Network Simulator, http://www.isi.edu/nsnam/ns

An Energy-Balanced Task Scheduling Heuristic for Heterogeneous Wireless Sensor Networks

Lee Kee Goh and Bharadwaj Veeravalli

Computer Networks and Distributed Systems Laboratory
Department of Electrical and Computer Engineering
National University of Singapore
4 Engineering Drive 3, Singapore 117576
{g0700296,elebv}@nus.edu.sg

Abstract. In this paper, we propose a static scheduling algorithm for assigning tasks with precedence constraints onto a cluster of heterogeneous sensor nodes connected by a single-hop wireless network so as to maximize the lifetime of the sensor network. The processing element on each sensor node is equipped with dynamic voltage scaling capability. In our algorithm, we assign the tasks to the sensor nodes so as to minimize the energy consumption of the tasks on each sensor node while keeping the energy consumption as balanced as possible. We also propose an algorithm to generate a second schedule that can improve the lifetime of the network further when it is used together with the original schedule. We observe up to 311% lifetime improvement in our simulations when our algorithms are compared to the baseline case where dynamic voltage scaling is not used.

Keywords: Energy-aware scheduling, power management, dynamic voltage scaling, heterogeneous multiprocessor scheduling, wireless sensor network.

1 Introduction

Today, wireless sensor networks (WSNs) are used in a wide variety of applications such as health monitoring, target tracking and surveillance. These applications often require each sensor node to sense and collect information from the surrounding environment, process the information collected, and communicate the results to other sensor nodes in the network. Based on the collective information gathered from several sensor nodes, a decision can then be made to determine the action to be taken.

In many WSNs, the sensors are individually operated by battery. Efficient energy management is required to prolong the lifetime of such sensor networks. While there are many energy-aware scheduling algorithms in the literature for both homogeneous [3, 4, 6, 7, 8, 10] and heterogeneous [5, 9, 11, 12, 13, 15] multiprocessor systems, these algorithms are designed mainly for tightly coupled systems and are not suitable for wireless sensor networks. These algorithms try

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 257–268, 2008.

to minimize the overall total energy consumption of the system in order to maximize its lifetime. This works for tightly coupled system since the processors in the system share the same energy source. However, for wireless sensor network, minimizing the total energy consumption does not necessarily maximize the lifetime of the network. For example, if many tasks are assigned to a single node, the battery on this node will be drained at a rate much faster than other nodes. As a result, the lifetime of the network is completely determined by the lifetime of this single node even when other nodes have an abundance of remaining energy. Therefore, in order to maximize the lifetime of a WSN, we should try to distribute the tasks among the sensor nodes in such a way that the energy consumption on each sensor node is as balanced as possible. Yu et al. [2] did so using a 3-phase heuristic approach. In the first phase, the tasks are grouped into clusters by eliminating communications with high execution times. Next, the clusters are assigned to the sensor nodes in a way such that the norm-energies of the sensor nodes are balanced. Here, the norm-energy is defined as the total energy consumption of the tasks scheduled on a node normalized by the remaining energy of that node. In the last phase, the voltage levels of the tasks are adjusted to reduce the energy consumption further. However, the 3-phase heuristic approach is only applicable to a WSN with homogeneous sensor nodes.

In this paper, we propose an energy-balanced task scheduling algorithm for assigning tasks with precedence constraints as represented by a task precedence graph onto a wireless network of heterogeneous sensor nodes. Each sensor node is equipped with dynamic voltage scaling (DVS) capability. In our algorithm, we try to minimize the maximum norm-energy among the sensor nodes during each step of assigning the tasks to the sensor nodes while ensuring that the deadline constraints will be met. We also try to minimize the total energy consumption for the whole network as well. In order to improve the lifetime of the network further, we propose another algorithm to generate a second schedule to be used together with the original schedule. We compare our algorithms to the baseline case where no DVS is used. We also apply our algorithms to a wireless network of homogeneous sensor nodes and compare their performance to the 3-phase heuristic [2].

The remainder of the paper is organized as follows: We introduce the system and task model in the next section. Sections 3 and 4 describe our proposed algorithms in detail. Simulation results and discussions are presented in Section 5. Lastly, we present our conclusions in Section 6.

2 Problem Formulation

2.1 System Model

Our system consists of a set of m heterogeneous sensor nodes, $\{PE_1, PE_2, \ldots, PE_m\}$, connected by a single-hop wireless network with K communication channels. Each sensor node is equipped with DVS functionality. Without loss of generality, we assume that each sensor node has D discrete voltage levels for simplicity. The computational speed of PE_i at voltage level V_j are given by S_{ij}.

The time cost and energy consumption for transmitting one unit of data between two sensor nodes PE_i and PE_j is denoted by τ_{ij} and ξ_{ij} respectively. We assume that the time and energy cost of wireless transmission is the same at both the sender and the receiver and no techniques such as modulation scaling [14] are used for energy-latency tradeoffs of communication activities. We also assume that negligible power is consumed by the sensor nodes and the radios when they are idle.

2.2 Task Model

We consider an application that is run periodically in the sensor network. Let P be the period of the application. Therefore, an instance of the application will be activated at time iP and it must be completed before the next instance is activated at time $(i+1)P$, where $i = 0, 1, 2, \ldots$. The application is represented by a directed acyclic task graph (DAG) $G = (T, E)$, which consists of a set of n dependent tasks $\{T_1, T_2, \ldots, T_n\}$ connected by a set of e edges $\{E_1, E_2, \ldots, E_e\}$. Each edge E_i from T_j to T_k has a weight C_i, which represents the number of units of data to be transmitted from T_j to T_k. The source tasks in G (i.e. tasks with no incoming edges) are used for measuring or collecting data from the environment and so they have to be assigned to different sensor nodes. We denote the time and energy cost of executing T_i on PE_j at the voltage level V_k by t_{ijk} and ϵ_{ijk} respectively. Let $\theta(T_i)$ denotes the sensor node to which T_i is assigned. The energy consumption of PE_i in one period of the application π_i is given by:

$$\pi_i = \sum_{j=1}^{n} \sum_{k=1}^{D} (x_{jk} \cdot \epsilon_{jik}) + \sum_{j=1}^{e} (y_j \cdot C_j \cdot \xi_{\theta(T_a)\theta(T_b)}) \tag{1}$$

where T_a and T_b are connected by the edge E_j and x_{jk} and y_j are defined as follows:

$$x_{jk} = \begin{cases} 1 \text{ if } T_j \text{ is scheduled on } PE_i \text{ at} \\ \quad \text{the voltage level } V_k \\ 0 \text{ otherwise} \end{cases}$$

$$y_j = \begin{cases} 1 \text{ if either } T_a \text{ or } T_b \text{ (but not} \\ \quad \text{both) is scheduled on } PE_i \\ 0 \text{ otherwise} \end{cases}$$

Let R_i be the remaining energy of PE_i. We define the norm-energy η_i [2] of PE_i as its energy consumption in one period normalized by its remaining energy:

$$\eta_i = \frac{\pi_i}{R_i} \tag{2}$$

The lifetime of the whole sensor network L is therefore determined by the sensor node with the largest norm-energy and is given by:

$$L = \lfloor \frac{1}{max_i(\eta_i)} \rfloor \tag{3}$$

3 Energy-Balanced Task Scheduling

In this section, we shall describe our Energy-Balanced Task Scheduling algorithm (EBTS) in detail. The pseudocode for our algorithm can be seen in Algorithm 1. In Step 1, we first calculate the average computational time of each task over all the sensor nodes and the average communication time of each edge over all pairs of sensor nodes. Next, we calculate the upper rank $rank_u$, as defined in [1], recursively using the following equation:

$$rank_u(T_i) = \overline{t_i} + \max_{T_j \in succ(T_i)} (\overline{\tau_{\theta(T_i)\theta(T_j)}} + rank_u(T_j)) \tag{4}$$

where $\overline{t_i}$ is the average computational time of T_i over all the sensor nodes, $\overline{\tau_{\theta(T_i)\theta(T_j)}}$ is the average communication time of the edge from T_i to T_j over all sensor node pairs and $succ(T_i)$ denotes the immediate successors of T_i.

In Step 2, the tasks are assigned in descending order of upper rank. For each task, we calculate the priority assuming that the task is assigned to each sensor node. We then assign the task to the sensor node that gives the lowest priority value, provided that its finish time when assigned to this sensor node does not exceed a threshold value $\sigma(T_i)$.

$$\sigma(T_i) = \frac{P}{m_s} \cdot f'_i \tag{5}$$

where m_s is the makespan of the schedule generated by a general list scheduling algorithm and f'_i is the finish time of T_i using the list scheduling algorithm. If the threshold is exceeded, the task will be assigned to the sensor node that gives the earliest finish time. We impose a threshold to the finish time to reduce the probability of many tasks being assigned to the same sensor node as this may result in deadlines to be missed.

In Step 3, we first identify the sensor node with the largest norm-energy. Among the tasks that are assigned to this sensor node, we select the one that has the largest energy reduction when assigned to the next voltage level without violating the deadline constraints. If no such tasks exists, the sensor node is removed permanently from the priority queue and does not need to be considered further. Otherwise, the new norm-energy of the sensor node is updated accordingly and the node is reinserted back into the priority queue. The process continues until no tasks can be executed at a lower voltage level without exceeding the deadline.

4 Energy-Balancing Using Two Schedules

Although our EBTS algorithm tries to improve the lifetime of the sensor network by balancing the energy consumption among the sensor nodes, it is often very difficult to obtain a perfectly energy-balanced schedule in practice. As a result, when the batteries of some sensor nodes are depleted, there is still a significant amount of remaining energy in other sensor nodes. If we can reassign the tasks

Algorithm 1. EBTS

```
Step 1: Calculate Upper Rank
Assign computation time of tasks with the mean values over all sensor nodes. Assign
  communication time of edges with mean values overall all pairs of sensor nodes.
Compute the upper rank of all the tasks by traversing the DAG upwards, starting
  from the sink tasks.

Step 2: Assign Tasks to Sensor Nodes
Sort the tasks in non-increasing order of upper rank.
while there are unscheduled tasks do
  Select the first task $T_i$ from the sorted list.
  for each sensor node $PE_j$ in the system do
    Assume that $T_i$ is allocated to $PE_j$.
    Compute the norm-energy $\eta_k$ for all sensor nodes where $k = 1, 2, \cdots m$.
    Compute the finish time $f_j$ of $T_i$.
    Calculate the priority function $pr_j = \alpha \cdot \max(\eta_k) + (1 - \alpha) \cdot \sum \eta_k$, where
      $\alpha \in [0, 1]$ is a user-defined parmeter. If $T_i$ is a source task and another source
      task is already assigned to $PE_j$, Set $pr_j = \infty$.
  end for
  Assign $T_i$ to the sensor node with the least value of $pr_j$, provided that the finish
    time $f_j$ of this assignment is less than or equal to the threshold $\sigma(T_i)$.
  If no such sensor node exists, assign $T_i$ to the sensor node that gives the smallest
    value of $f_j$.
end while

Step 3: Assign Voltage Levels
Insert the sensor nodes into a priority queue $Q$ sorted in non-increasing order of
  norm-energy.
while $Q$ is not empty do
  Remove the first sensor node $PE_j$ from $Q$.
  Sort the tasks assigned to this sensor node in non-increasing order of the differ-
    ence in energy consumption when their voltage is lowered by one level.
  for each task $T_i$ in the sorted list do
    Lower voltage of $T_i$ to the next lower voltage level.
    if deadline is not exceeded then
      Update the schedule.
      Insert this sensor node back into $Q$ with the updated value of norm-energy.
        Break out of for loop.
    end if
  end for
end while
```

from a sensor node that is low in remaining energy to one that is high in remaining energy in an effective way, the lifetime of the network can be increased further. In view of this, we propose to use two static schedules for executing the tasks instead of a single schedule. Our aim is to use a second schedule that compliments

Algorithm 2. EBTS-DS

```
Run the EBTS algorithm to obtain an initial schedule $S$ and the energy consumption per cycle $\pi_i$ of all sensor nodes, $\forall i = 1, 2, \cdots, m$.
Calculate the expected lifetime $L$ using Equation 3
Set the first schedule $S^{1st} \leftarrow S$, the energy consumption per cycle $\pi_i^{1st} \leftarrow \pi_i$ and the remaining energy capacity $R_i^{1st} \leftarrow R_i$, $\forall i = 1, 2, \cdots, m$.
Set $\delta \leftarrow \lfloor \frac{L}{10} \rfloor$.
for $j \leftarrow 1$ to 9 do
    Set $R_i \leftarrow (R_i^{1st} - j \times \delta \times \pi_i^{1st})$, $\forall i = 1, 2, \cdots, m$.
    Run the EBTS algorithm to schedule the non-source tasks using these values of $R_i$ as the remaining energies of the sensor nodes. Update $S$ and $\pi_i$ using the new schedule.
    if $S$ is feasible then
        Solve the following linear program:
        • Maximize $C_1 + C_2$
        • Subjected to $C_1 \times \pi_i^{1st} + C_2 \times \pi_i \leq R_i^{1st}$, $\forall i = 1, 2, \cdots, m$.
        if $(\lfloor C_1 \rfloor + \lfloor C_2 \rfloor) > L$ then
            Update $L \leftarrow (\lfloor C_1 \rfloor + \lfloor C_2 \rfloor)$.
            Update $\zeta^{1st} \leftarrow \lfloor C_1 \rfloor$ and $\zeta^{2nd} \leftarrow \lfloor C_2 \rfloor$, where $\zeta^{1st}$ and $\zeta^{2nd}$ are the number of cycles to run $S^{1st}$ and $S^{2nd}$ respectively.
            Update $S^{2nd} \leftarrow S$, $\pi_i^{2nd} \leftarrow \pi_i$ and $R_i^{2nd} \leftarrow R_i$, $\forall i = 1, 2, \cdots, m$.
        end if
    end if
end for
if $S^{2nd}$ is found then
    Use $S^{1st}$ for $\zeta^{1st}$ cycles followed by $S^{2nd}$ for $\zeta^{2nd}$ cycles.
else
    Use $S^{1st}$ for the entire lifetime of the sensor network.
end if
```

the first one such that the lifetime of the network can be increased by the combined use of both schedules compared to using each schedule individually. The EBTS with Dual Schedule (EBTS-DS) algorithm is shown in Algorithm 2.

In EBTS-DS, we first generate a schedule using our EBTS algorithm. This will be the first schedule S^{1st}. Next, we try to generate a series of candidate schedules. Each candidate schedule is generated in the following way. We assume that S^{1st} has been run for $\frac{j}{10}$ of its lifetime, where $j = 1, 2, \cdots, 9$. We then recalculate the remaining energy of each sensor node in the network. Based on the remaining energy, we reschedule the non-source tasks using our EBTS algorithm to obtain a new schedule. Since the source tasks are usually used to collect measurements from the environment, they are assigned to specific sensor nodes and cannot be reassigned. We then use a linear program to calculate the maximum possible lifetime that can be obtained using both the first schedule and the current candidate schedule. Lastly, we choose the candidate schedule that maximizes the network lifetime if it is used together with the first schedule.

5 Simulation

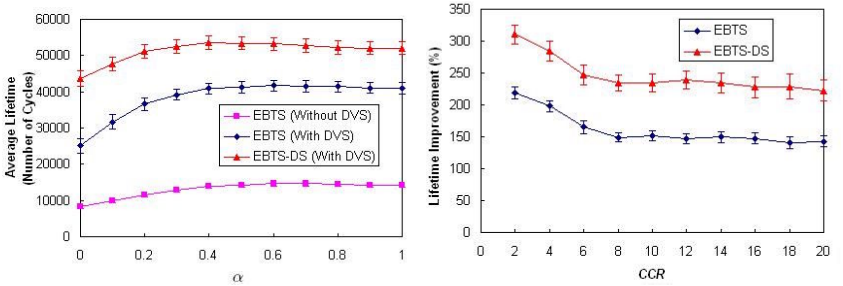
5.1 WSN with Heterogeneous Sensor Nodes

In this section, we describe the simulation study performed to evaluate the performance of our algorithms. First, we applied our algorithms to a WSN with heterogeneous nodes and compared their performance to the baseline case when EBTS is used without applying DVS. The algorithms were implemented using C++ in a Cygwin environment on a Pentium-IV /3.2GHz /2GB RAM PC running Windows XP. The task graphs used in the experiment were randomly generated using TGFF [16]. The maximum in-degree and out-degree of each task is set at 5. The number of source tasks in each task graph is equal to the number of sensor nodes used in the experiment. The average computation time and energy consumption of each task over all the sensor nodes at the maximum speed were randomly generated using a gamma distribution with a mean of 2msec and 4mJ respectively. The computation time and energy consumption of the task on each individual sensor node were then randomly generated using another gamma distribution with the mean equal to the average values generated earlier. We assumed that all the tasks executed at their worst-case execution time for every periodic cycle of the application. We also assumed that the minimum computational speed is $\frac{1}{N_v}$ of the maximum computational speed and all other levels of computational speed are uniformly distributed between the minimum and maximum speed.

We set the time and energy cost of transmitting one bit of data to be 10 μsec and 1 μJ respectively. The number of bits of data to be transmitted between 2 tasks with precedence constraints was uniformly distributed between $200CCR(1 \pm 0.2)$, where CCR represents the communication to computation ratio. The period of the application P was generated using the same method as described in [2] in the following way. For each task, its *distance* is defined as the number of edges in the longest path from any of the source tasks to that task. The tasks are then divided into layers according to their distance. Assuming that all the tasks in the same layer are executed in parallel, the estimated computation time required for each layer is $\overline{t_l}\lceil \frac{n_l}{m} \rceil$, where n_l is the number of tasks in that layer and $\overline{t_l}$ denotes the average computational time of the tasks in the layer. In addition, the expected number of communication activities initiated by any task is estimated to be $(d_{out} - 1)$ where d_{out} is the out-degree of the task. The total communication time required for each layer is therefore estimated as $\overline{t_l} \cdot CCR\lceil \frac{q}{K} \rceil$ where q is the sum of the expected number of communication activities initiated by the tasks in that layer. P is then calculated as the sum of computation and communication time of all the layers divided by u, where u is the utilization of the sensor network. Lastly, the remaining energy at each sensor node is uniformly generated between $(1 \pm 0.2) \times 10^6$mJ.

We conducted the simulation experiments for a wireless sensor network consisting of 10 sensor nodes, 8 voltage levels, 4 channels, 100 tasks (with 10 source tasks), CCR between 2 and 20, and u between 0 and 1. All the data in the experiment are obtained by averaging the values obtained using 100 different task graphs. We first set CCR to be 4, u to be 0.5 and varied the value of α from 0 to 1. The results are shown in Figure 1(a). We observed that when $\alpha = 0$, both our EBTS and EBTS-DS algorithms tried to minimize the total energy consumption of the sensor network without considering the maximum norm-energy of each sensor node. Therefore the average lifetime of the sensor network obtained using our algorithms are shorter. On the other hand, when α is around 0.4 to 0.6, the average lifetimes reach the maximum values before decreasing slightly when α increases further. This is because when α is around 0.4 to 0.6, our algorithms take into consideration the maximum norm-energy while trying to minimize the total energy consumption at the same time. As a result, the sensor nodes have more remaining energy and our algorithms are therefore able to generate better schedules. At $\alpha = 0.5$, there is about 198% improvement in the lifetime when EBTS is used with DVS, compared to the baseline case of using EBTS without DVS. When our EBTS-DS algorithm is used to generate a second schedule, the lifetime improvement increases to 290%. We shall use the value $\alpha = 0.5$ in our subsequent experiments.

Next, we varied CCR between 2 to 20. The lifetime improvement of our algorithms for different values of CCR is shown in Figure 1(b). Here, we define the *lifetime improvement* as $(\frac{L_{alg}}{L_{base}} - 1)$, where L_{alg} is the lifetime of the WSN when a particular algorithm is used and L_{base} is the lifetime of the WSN in the baseline case when our EBTS algorithm is used without DVS. When $CCR = 2$, our EBTS algorithm is able to obtain a lifetime improvement of about 219% while our EBTS-DS achieves a lifetime improvement of 311%. However, as CCR increases, this improvement decreases. At $CCR = 20$, the lifetime improvements that we could achieve using EBTS and EBTS-DS decrease to about 142% and 222% respectively. This is because as CCR increases, the communication energy becomes more significant when compared to the computational energy. Therefore,



(a) Average lifetime with varying α and $CCR = 4$

(b) Lifetime improvement with varying CCR and $\alpha = 0.5$

Fig. 1. Performance of EBTS and EBTS-DS for WSN consisting of heterogeneous nodes (10 sensor nodes, 8 voltage levels, 4 channels, 100 tasks, $u = 0.5$). The values for the lifetime improvement are calculated as the improvement over the baseline case when EBTS is used without DVS. The vertical bars show the confidence intervals at 95% confidence level.

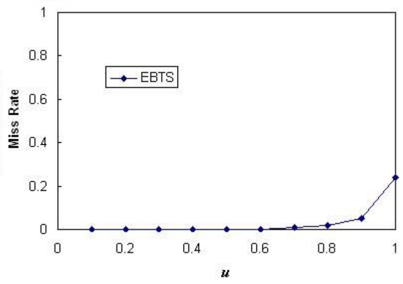


Fig. 2. Miss rate of EBTS with varying values of u for WSN consisting of heterogeneous nodes (10 sensor nodes, 8 voltage levels, 4 channels, 100 tasks, $CCR = 0$)

the lifetime improvement obtained by reducing the computational energy using DVS becomes more limited as a result.

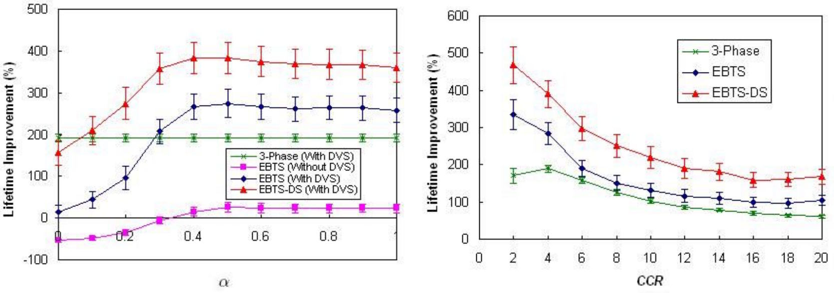
We now varied the utilization u of the sensor network and observed the rate at which deadlines were missed. Figure 2 shows the simulation results. We observed that our algorithm provides a very low miss rate. Even when $u = 1$, only 24% of the application task graphs missed their deadlines.

5.2 WSN with Homogeneous Sensor Nodes

We also compared our algorithms with the 3-phase heuristic [2] for WSNs with homogeneous nodes. The task graphs and parameters are generated in the same way as in the previous experiment, except that the computational time and energy consumption for each task does not vary across different sensor nodes. We compared the performance of the algorithms to the baseline case when the 3-phase heuristic is used without applying DVS.

Figure 3(a) shows the results when α is varied from 0 to 1. We observed similar results as in the previous experiment. When $\alpha = 0$, the lifetime improvements of EBTS and EBTS-DS are only 15% and 156% respectively compared to 192% improvement obtained using the 3-phase heuristic. On the other hand, when $\alpha = 1$, lifetime improvement of EBTS and EBTS-DS are 258% and 360% respectively. The best performance is obtained when $\alpha = 0.5$. At this value of α, the lifetime improvement of EBTS and EBTS-DS are 275% and 383% respectively. Even when our EBTS algorithm is used without DVS, there is an improvement of 25% at $\alpha = 0.5$ compared to the baseline case of using the 3-phase heuristic without DVS.

Next, we studied the performance of the algorithms with respect to varying values of CCR. The results are shown in Figure 3(b). The lifetime improvement is calculated by comparing the lifetime generated by the algorithms to the baseline case where the 3-phase heuristic is used without DVS. When $CCR = 2$, our EBTS and EBTS-DS algorithms obtain lifetime improvements of 335% and 468% respectively while the 3-phase heuristic obtains an improvement of only 171%.



(a) Lifetime improvement with varying α and $CCR = 4$

(b) Lifetime improvement with varying CCR and $\alpha = 0.5$

Fig. 3. Lifetime improvement of 3-phase heuristic, EBTS and EBTS-DS for WSN consisting of homogeneous nodes (10 sensor nodes, 8 voltage levels, 4 channels, 100 tasks, $u = 0.5$). These values are calculated as the improvement over the baseline case when the 3-phase heuristic is used without DVS. The vertical bars show the confidence intervals at 95% confidence level.

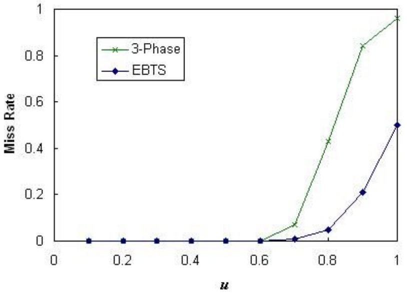


Fig. 4. Miss rate of 3-phase heuristic and EBTS with varying values of u for WSN consisting of homogeneous nodes (10 sensor nodes, 8 voltage levels, 4 channels, 100 tasks, $CCR = 0$)

However, as CCR increases, the improvement of EBTS and EBTS-DS over the 3-phase heuristic decreases. At $CCR = 20$, EBTS and EBTS-DS achieve lifetime improvements of about 105% and 168% respectively compared to 60% achieved by the 3-phase heuristic.

Lastly, we studied the deadline miss rates of our EBTS algorithm and the 3-phase heuristic for homogeneous WSNs. The results are shown in Figure 4. We observed that our EBTS algorithm provides a lower deadline miss rate compared to the 3-phase heuristic, especially at higher values of u. At $u = 1$, the miss rate of our algorithm is 50% while the miss rate for the 3-phase heuristic is as high as 96%. From these experiments, we conclude that although our algorithms are designed for WSNs with heterogeneous sensor nodes, they can be used for homogeneous sensor nodes as well.

6 Conclusion

In this paper, we have proposed a static energy-balanced task scheduling (EBTS) algorithm for assigning tasks with precedence constraints to a single-hop WSN consisting of heterogeneous nodes. Our objective is to maximize the lifetime of the WSN by assigning the tasks in a balanced way such that the lifetime of the sensor node which consumes the most energy is maximized. We also proposed the EBTS-DS algorithm, which generates a second schedule that is used to extend the lifetime of the WSN further when it is used together with the original schedule.

We compared our EBTS and EBTS-DS algorithms to the baseline case when DVS was not used for a WSN with heterogeneous nodes and observed up to 219% and 311% improvement in the lifetime of the sensor network respectively. We also compared our EBTS and EBTS-DS algorithms to the 3-phase heuristic in the literature for sensor networks with homogeneous nodes and demonstrated that there is up to 96% and 174% increase in the lifetime improvement respectively when our algorithms are used instead of the 3-phase heuristic.

Acknowledgements

Bharadwaj V would like to acknowledge the funding support for this project from Incentive Funding scheme by Faculty of Engineering, NUS Singapore, Grant #R-263-000-375-731. Parts of this work also contribute to a funded project (R-263-000-375-305) from A*STAR EHS II Programme.

References

1. Baskiyar, S., Palli, K.K.: Low Power Scheduling of DAGs to Minimize Finish Times. In: Robert, Y., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2006. LNCS, vol. 4297, pp. 353–362. Springer, Heidelberg (2006)
2. Yu, Y., Prasanna, V.K.: Energy-Balanced Task Allocation for Parallel Processing in Wireless Sensor Networks. Mobile Networks and Applications 10, 115–131 (2005)
3. Han, J.-J., Li, Q.-H.: Dynamic Power-Aware Scheduling Algorithms for Real-Time Task Sets with Fault-Tolerance in Parallel and Distributed Computing Environment. In: Proceedings of 19th International Parallel and Distributed Processing Symposium (April 2005)
4. AlEnawy, T.A., Aydin, H.: Energy-Aware Task Allocation for Rate Monotonic Scheduling. In: Proceedings of 11th IEEE Real-Time and Embedded Technology and Applications Symposium, March 2005, pp. 213–223 (2005)
5. Gorji-Ara, B., Chou, P., Bagherzadeh, N., Reshadi, M., Jensen, D.: Fast and Efficient Voltage Scheduling by Evolutionary Slack Distribution. In: Proceedings of Asia and South Pacific Design Automation Conference, January 2004, pp. 659–662 (2004)
6. Zhu, D., Melhem, R.G., Childers, B.R.: Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multiprocessor Real-Time Systems. IEEE Trans. Parallel and Distributed Systems 14(7), 686–700 (2003)

7. Aydin, H., Yang, Q.: Energy-Aware Partitioning for Multiprocessor Real-Time Systems. In: Proceedings of 17th International Parallel and Distributed Processing Symposium (April 2003)
8. Mishra, R., Rastogi, N., Zhu, D., Mossé, D., Melhem, R.G.: Energy Aware Scheduling for Distributed Real-Time Systems. In: Proceedings of 17th International Parallel and Distributed Processing Symposium (April 2003)
9. Yu, Y., Prasanna, V.K.: Power-Aware Resource Allocation for Independent Tasks in Heterogeneous Real-Time Systems. In: Proceedings of 9th International Conference on Parallel and Distributed Systems, December 2002, pp. 341–348 (2002)
10. Zhu, D., AbouGhazaleh, N., Mossé, D., Melhem, R.G.: Power Aware Scheduling for AND/OR Graphs in Multi-Processor Real-Time Systems. In: Proceedings of 31st International Conference on Parallel Processing, August 2002, pp. 593–601 (2002)
11. Schmitz, M.T., Al-Hashimi, B.M., Eles, P.: Energy-Efficient Mapping and Scheduling for DVS Enabled Distributed Embedded Systems. In: Proceedings of 2002 Design, Automation and Test in Europe Conference and Exposition, March 2002, pp. 514–521 (2002)
12. Schmitz, M.T., Al-Hashimi, B.M.: Considering Power Variations of DVS Processing Elements for Energy Minimisation in Distributed Systems. In: Proceedings of International Symposium on Systems Synthesis, October 2001, pp. 250–255 (2001)
13. Gruian, F., Kuchcinski, K.: LEneS: Task Scheduling for Low-Energy Systems Using Variable Supply Voltage Processors. In: Proceedings of Asia and South Pacific Design Automation Conference, January 2001, pp. 449–455 (2001)
14. Schurgers, C., Aberthorne, O., Srivastava, M.B.: Modulation Scaling for Energy-aware Communication Systems. In: Proceedings of International Symposium on Low Power Electronics and Design, pp. 96–99 (2001)
15. Luo, J., Jha, N.K.: Power-conscious Joint Scheduling of Periodic Task Graphs and Aperiodic Tasks in Distributed Real-time Embedded Systems. In: Proceedings of International Conference on Computer-Aided Design, November 2000, pp. 357–364 (2000)
16. Dick, R.P., Rhodes, D.L., Wolf, W.: TGFF: Task Graphs for Free. In: Proceedings of 6th International Workshop on Hardware/Software Codesign, March 1998, pp. 97–101 (1998)

Energy Efficient Distributed Algorithms for Sensor Target Coverage Based on Properties of an Optimal Schedule

Akshaye Dhawan and Sushil K. Prasad

Georgia State University, Department of Computer Science, Atlanta, Ga 30303
akshaye@cs.gsu.edu, sprasad@cs.gsu.edu

Abstract. A major challenge in Wireless Sensor Networks is that of maximizing the lifetime while maintaining coverage of a set of targets, a known NP-complete problem. In this paper, we present theoretically-grounded, energy-efficient, distributed algorithms that enable sensors to schedule themselves into sleep-sense cycles. We had earlier introduced a lifetime dependency (LD) graph model that captures the interdependencies between these cover sets by modeling each cover as a node and having the edges represent shared sensors. The key motivation behind our approach in this paper has been to start with the question of what an optimal schedule would do with the lifetime dependency graph. We prove some basic properties of the optimal schedule that relate to the LD graph. Based on these properties, we have designed algorithms which choose the covers that exhibit these optimal schedule like properties. We present three new sophisticated algorithms to prioritize covers in the dependency graph and simulate their performance against state-of-art algorithms. The net effect of the 1-hop version of these three algorithms is a lifetime improvement of more than 25-30% over the competing algorithms of other groups, and 10-15% over our own; the 2-hop versions have additional improvements, 30-35% and 20-25%, respectively.

1 Introduction

Wireless Sensor Networks (WSNs) consist of a number of low cost sensors that are equipped with a radio interface. These devices are deployed in large numbers over an area of interest and they monitor the targets in this region and send information to a base station or a gateway node [1].

Since these sensors are powered by batteries, energy is a key constraint for these networks. The lifetime of the network is defined as the amount of time that the network can cover its *area* or *targets* of interest. A standard approach taken to maximize the lifetime is to make use of the overlap in the sensing regions of individual sensors caused by the high density of deployment. Hence, only a subset of all sensors need to be in the "on" state, while the other sensors can enter a low power "sleep" state. The members of this active *cover* set, are then periodically updated. In using such a scheduling scheme, there are two problems that need to be addressed. First, we need to determine how long to use a given

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 269–281, 2008.

cover set and then we need to decide which set to use next. This problem has been shown to be NP-complete [2, 3].

Existing work on this problem has looked at both centralized and distributed algorithms to come up with such a schedule. Initial approaches to maximize the lifetime in [2, 3, 4] considered the problem of finding the maximum number of *disjoint* covers. However, [5, 6] and others showed that using non-disjoint covers allows the lifetime to be extended further and this has been adopted since. A common approach taken with centralized algorithms is that of formulating the problem as an optimization problem and using linear programming (LP) to solve it [3, 6, 7, 8]. The distributed algorithms typically operate in rounds. At the beginning of each round, a sensor exchanges information with its neighbors, and makes a decision to either switch on or go to sleep. In most greedy algorithms [2, 4, 5, 9, 10, 11], the sensor with some simple greedy criteria like the largest uncovered area [9], maximum uncovered targets [10], etc. is selected to be on.

A key problem here is that since a sensor can be a part of multiple covers, these covers have an impact on each other, as using one cover set reduces the lifetime of another set that has sensors common with it. By making greedy choices, the impact of this dependency is not being considered. In [12], we capture this dependency between covers by introducing the concept of a local Lifetime Dependency (LD) Graph. This consists of the cover sets as nodes with any two nodes connected if the corresponding covers intersect (See Section 2 for a more detailed description). We also presented a generalized framework based on this approach and applied it to the area and k-coverage problems [13].

Our Contributions: The key motivation behind our approach in this paper has been to start with the question of what an optimal schedule (henceforth called *OPT*) would do with the LD graph. We have been able to prove certain basic properties of the *OPT* schedule that relate to the LD graph. Based on these properties, we have designed algorithms which choose the covers that exhibit these *OPT* schedule like properties. We present three new heuristics - *Sparse-OPT* based on the sparseness of connectivity among covers in *OPT*, *Bottleneck-Target* based on selecting covers that optimize the use of sensors covering local bottleneck targets, and *Even-Target-Rate* based on trying to achieve an even burning rate for all targets. These heuristics are at a higher level and operate on top of degree-based heuristics to prioritize the local covers. Our experiments show an improvement in lifetime of 10-15% over our previous work in [12] and 25-30% over competing work in [10, 11] and 35% improvement for a two-hop version over the two-hop algorithm of [11].

The remainder of this paper is organized as follows. We begin by defining the notation we use and review the Lifetime Dependency Graph model in Section 2. In Section 3 we present and prove some key properties of the *OPT* sequence. This is followed by Section 4 where we introduce new algorithms based on these properties. All the proposed heuristics are then simulated and evaluated in Section 5. Finally, we conclude in Section 6.

2 Preliminaries and Background Work

Definitions: Let the sensor network be represented using graph SN where, $S = \{s_1, s_2, \ldots, s_n\}$ is the set of sensors, and an edge between sensor s_i and s_j exists if the two sensors are in communication range of each other. Let the set of targets be $T = \{t_1, t_2, \ldots, t_m\}$. In this paper we consider the problem of covering a stationary set of targets. This can easily be translated into the area coverage problem by mapping the area to a set of points which need to be covered [4, 14]. In addition to this, we define the following notation:

- $b(s)$: The battery available at a sensor s.
- $T(s)$: The set of targets in the sensing range of sensor s.
- $N(s, k)$: The closed neighbor set of sensor s at k communication hops.
- Cover C: Cover $C \subseteq S$ to monitor targets in T is a minimal set of sensors s.t. each target $t \in T$ has a nearby sensor $s \in C$ which can sense t, i.e., $t \in T(s)$.
- $lt(C)$: Maximum lifetime of a cover C is $lt(C) = min_{s \in C} b(s)$.
- $lt(t_i)$: The lifetime of a target $t_i \in T$ is given by $lt(t_i) = \sum_{\{s | t_i \in T(s)\}} b(s)$.
- Bottleneck Sensor: Bottleneck sensor s of cover C is the sensor $s \in C$ with minimum battery, i.e., it is the sensor s that upper bounds $lt(C)$.
- Bottleneck Target (t_{bot}): The target with the smallest lifetime $lt(t_{bot})$.
- Lifetime of a schedule of covers: We can view the set of currently active sensors as a cover C_i that is used for some length of time l_i.Given a schedule of covers of the form, $(C_1, l_1), (C_2, l_2), \ldots, (C_r, l_r)$. The lifetime of this schedule is given by $\sum_{i=1}^{r} l_i$.
- OPT: The optimal schedule of covers that achieves the maximum lifetime. Note that this includes both the covers and their corresponding time periods.

The Lifetime Dependency (LD) Graph [12]: The Lifetime dependency graph $LD = (V, E)$ where V is the set of all possible covers to monitor targets in T and two covers C and C' are joined by an edge in E if and only if $C \cap C' \neq \emptyset$.

The LD graph effectively captures the dependency between two cover sets by representing their intersection by the edge between them. Further, we define,

- $w(e)$: Weight of an edge e between covers C and C' is $w(e) = min_{s \in C \cap C'} b(s)$.
- $d(C)$: Degree of a node or cover C is $d(C) = \Sigma_{e \ incident \ to \ C} \ w(e)$.

The reasoning behind this definition of the edge weight comes from considering a simple two-node LD graph with two covers C_1 and C_2 sharing an edge e. The lifetime of the graph is upper bounded by $min(lt(C_1) + lt(C_2), w(e))$. Similarly, in defining the degree of a cover C by summing the weights of all the edges incident on the cover, we are getting a measure of the impact it would have on all other covers with which it shares an edge.

Basic Algorithmic Framework: Our distributed algorithms consist of a initial setup phase followed by rounds of predetermined duration during which sensors negotiate with their neighbors to determine their sense/sleep status.

Setup: In the setup phase, each sensor s communicates with each of its neighbor $s' \in N(s, 1)$ exchanging battery levels $b(s)$ and $b(s')$, and the targets covered $T(s)$

and $T(s')$. Then it finds all the local covers using the sensors in $N(s, 1)$ for the target set being considered. It then constructs the local LD graph $LD = (V, E)$ over those covers, and calculates the degree $d(C)$ of each cover $C \in V$ in the graph LD. Note that the maximum number of covers that each sensor constructs is a function of the number of neighbors and the number of local targets it has. Both of these are relatively small for most graphs (but theoretically is exponential in the number of targets).

Prioritize and Negotiate solutions: Once the LD graph has been constructed by each sensor, it needs to decide which cover to use. In order to do this, a *priority function* can be defined to prioritize the local covers. We base the priority of cover C on its degree *d(C)*. A lower degree is better since this corresponds to a smaller impact on other covers. Note that the priority function is computed at the beginning of every round by exchanging current battery levels among neighbors since the degrees may have changed from the previous round.

The goal is to try and satisfy the highest priority cover. However, a cover comprises of multiple sensors and if one of these switches off, this cover cannot be satisfied. Hence, each sensor now uses the automaton in Fig. 1 to decide whether it can switch off or if it needs to remain on. The automaton starts with every sensor s in its highest priority cover C. The sensor s keeps trying to satisfy this cover C and eventually if the cover C is satisfied, then s switches on if $s \in C$ else s switches off. If a cover C cannot be satisfied, then the sensor s transitions to its next best priority cover C', C'' and so on, until a cover is satisfied.

We simulated this Degree-Based heuristic along with a few of its variants over a range of sensor networks and compared the lifetime of their schedules with the current state-of-art algorithms, LBP [10] and DEEPS [11], and showed an improvement of 10-20% in network lifetime. One clear distinction between our algorithm and others is that while all previous algorithms work on sensor network graph (SN), ours work on dependency graphs (LD), a higher level abstraction.

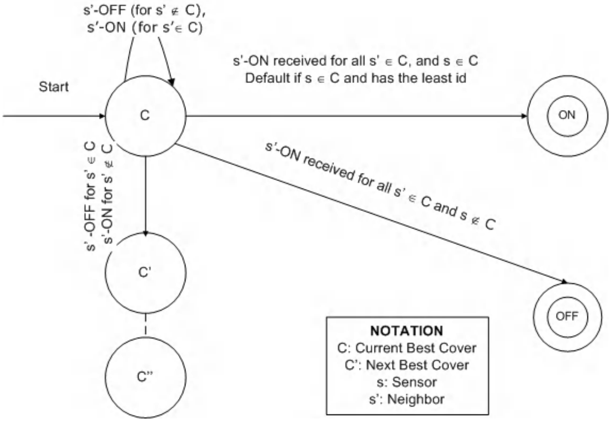


Fig. 1. The state transitions to decide the sense/sleep status

3 Properties of the Optimal Sequence

In this paper, our approach to the problem of further extending the lifetime of the network has been to turn the argument on its head. Instead of trying to come up with different schemes of scheduling sensors into cover sets, let us suppose we know the best schedule. Such a schedule OPT comprises a collection of covers and the time for which each cover is used in the optimal schedule,

$$OPT = \{(C_{opt_1}, l_1), (C_{opt_2}, l_2),\ ...,\ (C_{opt_r}, l_r)\}.$$

The optimal sequence can be viewed as a partition of the space of covers into those covers that are in OPT and those that are not. Now visualize all possible cover sets in the LD graph. How would one identify from amongst all these sets, those that are candidates for being in OPT? Our goal is to identify certain properties of the OPT that can help make this decision.

Lemma 1. OPT burns all the covers.

Proof: Suppose there is a cover C' that is not burned by OPT, i.e., its weakest sensor has some battery left. Burning this cover would further increase the lifetime of the network by an amount $lt(C')$. This implies that OPT is non-optimal, which gives us a contradiction. □

Lemma 2. If a cover C is not used in OPT, C has at least one neighboring cover in the LD Graph in OPT.

Proof: If a cover C is not used in OPT, then it has at least one sensor $s \in C$ that has exhausted its battery. If this is not true, then C still has some lifetime left and this gives us a contradiction according to Lemma 1. The fact that sensor s has been completely burned for cover C implies that there is at least one other cover in OPT that contains this sensor s. Otherwise, s would not be burned. This means that there is at least one cover which is a neighbor of C in the LD graph (because they share the sensor s) and is also in OPT. □

Corollary: If a cover C is not used in OPT, C has one or more neighboring covers in OPT such that total span of these neighbors in OPT is at least the life of C.

Lemma 3. The covers in OPT form a dominating set of the LD Graph.

Proof: A dominating set of a graph is a set of vertices such that every vertex is either in this set or is a neighbor of a node in the dominating set.

Let us consider a cover C. Either it is in OPT, in which case it is a dominating node, or it is not in OPT. By Lemma 2, if it is not in OPT, it has to have at least one neighbor in OPT. Hence it is a dominated node. Hence, the covers in OPT dominate the set of all covers in the LD graph. □

Lemma 4. All permutations of OPT are optimal.

Proof: This lemma shows that the ordering of individual covers in the OPT sequence is irrelevant. Any permutation OPT' of OPT can be obtained by repeatedly moving (C_{OPT_i}, l_i) to its position j in OPT' for all eligible i. We prove that each such move preserves optimality.

Moving C_{OPT_i} to position j changes its relative ordering with its neighboring covers in the LD graph which lie between position i and j. Let us call these neighbors $N'(C_{OPT_i})$. Other neighbors of C_{OPT_i} do not see any change, nor do other covers which are not neighbors. For each neighboring cover $C_{OPT_k} \in N'(C_{OPT_i})$, $w(C_{OPT_i} \cap C_{OPT_k}) \geq l_i + l_k$, as each edge upper bounds the cumulative lifetime. Therefore, if $i > j$ burning C_{OPT_i} before C_{OPT_k} will leave $w(C_{OPT_i} \cap C_{OPT_k}) - l_i \geq l_k$ of battery in $C_{OPT_i} \cap C_{OPT_k}$. Therefore, C_{OPT_k} can be burnt for a duration of l_k, for each $C_{OPT_k} \in N'(C_{OPT_i})$.

On the other hand, if $i < j$, each $C_{OPT_k} \in N'(C_{OPT_i})$ will be burnt for l_k time, leaving $w(C_{OPT_k} \cap C_{OPT_i}) - l_k \geq l_i$ of battery in $C_{OPT_k} \cap C_{OPT_i}$. Thus, C_{OPT_i} can be burnt for duration of l_i at position j. □

Corollary: If C occurs more than once in OPT, all its occurrences can be brought together, thereby burning C all at once for the cumulative duration.

Lemma 5. Due to OPT, all sensors around at least one target are completely burnt.

Proof: This relates to Lemma 1, because if there is no such bottleneck target, then it is still possible to cover all targets, implying that a cover exists, and hence OPT is not optimal. □

4 Optimal Schedule Based Algorithms

Recall from our discussion on the Lifetime Dependency Graph model in Section 2 that we used the degree $d(C)$ of a cover C in order to determine its priority. The definition of the degree as the key prioritization criteria is limited since it only considers a local property of the LD graph. As we saw in the previous section, there are properties which show how an OPT schedule would choose covers in the LD graph. Our goal in this section is to design heuristics that utilize these properties in the prioritization phase of the algorithm described in Section 2. We introduce three heuristics based on the properties of the OPT schedule. Each of these heuristics define a different way to prioritize the local covers. Note that if all remaining covers are tied on the new priority functions, we revert to using the degree to break ties. Hence, these heuristics can be viewed as defining a higher level priority function, that acts on top of the degree $d(C)$ function.

Heuristic 1: Sparse-OPT

This heuristic is based on *Lemma* 2 and works on the idea that the covers in OPT are sparsely connected. Suppose we have a subsequence of OPT available and we were to pick a cover to add to this sequence. Clearly any cover we add to the OPT sequence should not be a neighbor of a cover that is already in OPT. By *Lemma* 2, we know that if a cover C is in OPT for time l, then its neighbors in the LD graph can only be in OPT if their shared sensor s has a battery $b(s) > l$. This implies that covers in OPT are likely to be sparsely connected. Hence, for any two nodes (covers) in OPT, their degree in the induced subgraph of the LD graph should be low. For a cover C, we define its degree to other covers already selected (in OPT) as: $d_{OPT}(C) = \Sigma_{e\ incident\ to\ C\ and\ to\ C' \in OPT}\ w(e)$.

This leads us to a simple heuristic: *When choosing a cover C_i, pick the cover with the lowest degree to nodes already chosen.*

Implementation: When implementing this heuristic, the exchange of messages during the setup phase remains unchanged from before. The next step in the algorithm would be to prioritize the covers in the local LD graph. Instead of prioritizing a cover C by its degree $d(C)$, the heuristic we defined gives us a higher level of prioritization. Initially, there are no covers that have been selected for use so the heuristic starts off as before. A sensor orders its local covers by $d(C)$ (battery and id's can be used to break ties), and then enters a *negotiation* phase with its neighbors in order to decide which cover to use. This would result in some cover C' being selected for use.

In the subsequent round the sensor recomputes the priority function for its covers. Now, in computing the priority of a cover C, we can look at its degree to the previously used cover C', given by $d_{OPT}(C)$, and prioritize the covers in the order of lowest degree first. Note that if d_{OPT} is the same for several covers, we can break ties by using $d(C)$ as before. As the set of previously used covers increases over time, we compute the priorities at the beginning of each round by looking at the d_{OPT} of any cover to this set and assigning a higher priority to the covers with the lowest degree.

Heuristic 2: Bottleneck-Target

This heuristic is based on the property that covers in OPT should optimize the local bottleneck target and makes use of the ideas presented in *Lemma* 5.

Let us consider the set of all targets T. Some of these targets are more important than others because they represent bottlenecks for the lifetime of the network. Consider a target t_i. Then, the total amount of time this target can be monitored by any schedule is given by: $lt(t_i) = \sum_{\{s\ \mid\ t_i \in T(s)\}} b(s)$.

Clearly there is one such target with the smallest $lt(t_i)$ value, that is a bottleneck for the entire network. Since the network as a whole cannot achieve a lifetime better than this bottleneck, it follows that any cover should not use multiple sensors from the set of sensors that cover this bottleneck. However, without a global picture, it is not possible for any sensor to determine if one of its local targets is this global bottleneck. However, every sensor is aware of which of its local targets is a local bottleneck. The key thing to realize is the fact that if every sensor optimizes its local bottleneck target, then one of these local optimizations is also optimizing the global bottleneck target. Let t_{bot} be this local bottleneck target. Let C_{bot} be the set of sensors that can cover this local bottleneck target.That is, $C_{bot} = \{s \mid t_{bot} \in T(s)\}$.

Ideally, we would like to use a cover that has only one sensor in C_{bot} as a part of this cover. However this may not always be possible. Hence, while prioritizing the local covers, we can pick a cover that minimizes the cardinality of its intersection with C_{bot}. *Hence, the cover selected should be the cover C that minimizes $|C \cap C_{bot}|$.*

Implementation: The implementation is similar to Heuristic 1. Again, the setup and LD graph construction phase remain unchanged. Every sensor s now computes its local bottleneck target $t_{bot} \in T(s)$ and the set of sensors that covers this target C_{bot}. Note that this can be done since at the end of the setup phase, every sensor knows who its neighbors are, which targets they cover, and how much battery they have. When calculating the priority of each cover C in the LD graph, the priority function defined by this heuristic calculates the value of $|C \cap C_{bot}|$ for each cover, since this gives us the number of sensors in C_{bot} that are a part of the cover C. The heuristic then prioritizes covers in descending order of this cardinality. Again, if this value is the same for multiple covers, we break ties by using the degree $d(C)$.

Heuristic 3: Even-Target-Rate
This heuristic is based on the idea that OPT should try and burn all targets at the same rate. Consider a target $t_i \in T$. Let $lt(t_i)$ be the sum of the battery of all sensors covering t_i. Clearly the network cannot be alive for longer than $lt(t_{bot})$ where t_{bot} is the target with the smallest lifetime. Heuristic 2 stated above tries to maximize the time this bottleneck can be used. However, the danger in this is that by doing so, a different target may become the bottleneck due to repeated selection of its covering sensors. To avoid this problem, and at the same time optimize the bottleneck target, we want to keep an even rate of burning for all targets. In order to arrive at a normalized burning rate for each target t_i, we define the impact of a cover C on a target t_i as given by, $Impact(C, t_i) = \frac{|\{\ s \in C \mid t_i \in T(s)\}|}{lt(t_i)}$.

The *Impact* should give a measure of how this cover C is reducing the lifetime of a target t_i. Since each round lasts for one time unit, the impact is measured by the number of sensors covering t_i that are in C. Hence, the definition. This gives us the heuristic: *Any cover chosen should be the best fitting cover in that it burns all targets at an even rate.*

Implementation: After setup and constructing its LD Graph as before, each sensor s enters the prioritization phase. For every target being considered, s computes the impact of a cover C on that target, i.e., the sensor s calculates $Impact(C, t_i)$ for all $t_i \in T(s)$. Let $Impact_{max}(C)$ be the highest impact of this cover C and let $Impact_{min}(C)$ be the lowest impact of C for all targets. A good cover will have a similar impact on all targets, since it burns them at the same normalized rate. Hence, we prioritize covers in descending order of this difference, given by $Impact_{max}(C) - Impact_{min}(C)$. Once again, if the impact is the same, we can break ties using $d(C)$.

5 Experimental Evaluation

In this section, we first evaluate the performance of the one-hop and two-hop versions of the three heuristics based on OPT schedule as compared to the 1-hop algorithm LBP [10], the 2-hop algorithm DEEPS [11], and our basic Degree-Based heuristic from our previous work in [12]. Next, we create appropriate hybrids of the three 1-hop heuristics in various combinations, showing that all three combined together yield around 30% improvement over 1-hop LBP and 25% over the 2-hop DEEPS algorithm (Fig. 2 and 3). Even though theoretically our algorithms are exponential in the number of targets, practically their computation time was no more than three times the LBP algorithm with same communication complexities. We also implement the 2-hop versions of these heuristics and obtain a 35% improvement in lifetime over DEEPS (Fig. 4). As compared to an upper bound on the longest network lifetime (the lifetime of the global bottleneck target), our 1-hop algorithms have moved the state of art to no worse than 30-40% on an average (Fig. 5). The 2-hop algorithms are no worse that 25% on an average.

The load balancing protocol (LBP) [10] is a simple one-hop protocol which works by attempting to balance the load between sensors. Sensors can be in one of three states sense, sleep or vulnerable/undecided. Initially all sensors are vulnerable and broadcast their battery levels along with information on which

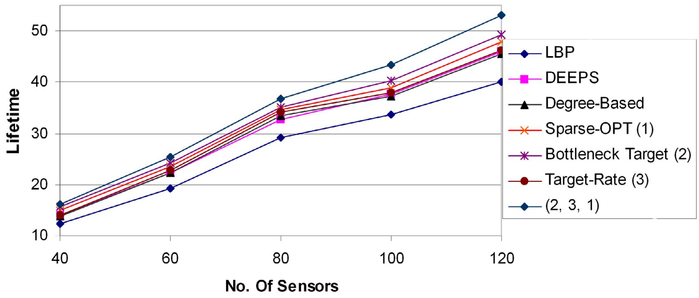


Fig. 2. Lifetime with 25 Targets

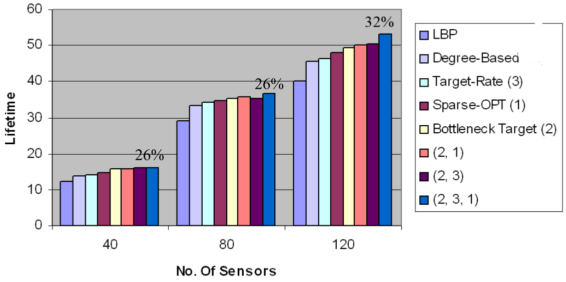


Fig. 3. Comparing LBP [3] against the 1-hop OPT-based heuristics

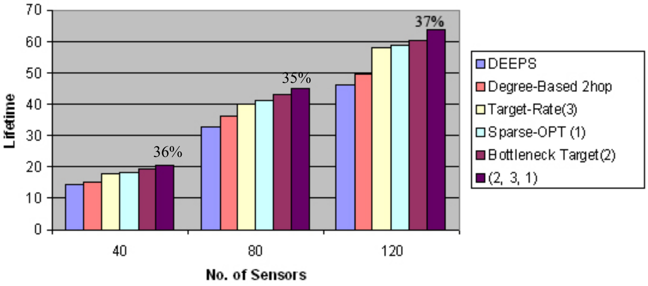


Fig. 4. Comparing DEEPS [11] against the 2-hop OPT-based heuristics

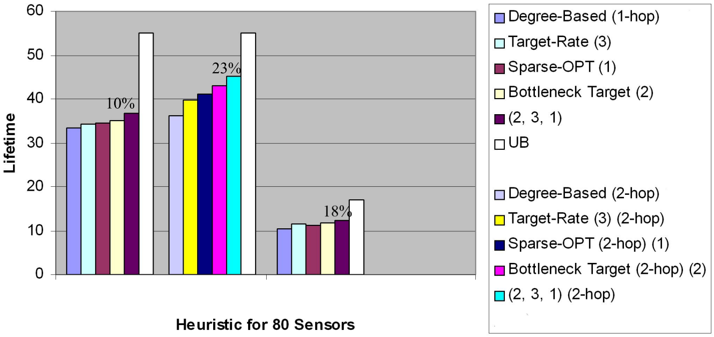


Fig. 5. Comparing 1-hop and 2-hop Degree-Based [12] heuristics against OPT-based heuristics

targets they cover. Based on this, a sensor decides to switch to off state if its targets are covered by a higher energy sensor in either on or vulnerable state. On the other hand, it remains on if it is the sole sensor covering a target.

The second protocol we consider is a two-hop protocol called DEEPS [11].The main intuition behind DEEPS is to try to minimize the energy consumption rate around those targets with smaller lives. A target is defined as a *sink* if it is the shortest-life target for at least one sensor covering that target. Otherwise, it is a *hill*. To guard against leaving a target uncovered during a shuffle, each target is assigned an in-charge sensor. For each sink, its in-charge sensor is the one with the largest battery for which this is the shortest-life target. For a hill target, its in-charge is that neighboring sensor whose shortest-life target has the longest life. An in-charge sensor does not switch off unless its targets are covered by someone. Apart from this, the rules are identical as those in LBP protocol.

In order to compare the algorithm against LBP, DEEPS, and our previous work, we use the same experimental setup and parameters as employed in [10]. We carry out all the simulations using $C{+}{+}$. For the simulation environment, a static wireless network of sensors and targets scattered randomly in $100m \times 100m$ area is considered. We conduct the simulation with 25 targets randomly deployed, and vary the number of sensors between 40 and 120 with an increment of 20 and each sensor with a fixed sensing range of $60m$. We assume that the communication range of each sensor is two times the sensing range [15, 16]. For these simulations, we use the linear energy model wherein the power required to sense a target at distance d is proportional to d. We also experimented with the quadratic energy model (power proportional to d^2). The results are similar to [12] and the same trends as the linear model apply. Due to space constraints, we only show a snapshot of these results in Fig. 5.

The results are shown in Fig. 2. As can be seen from the figure, among the three heuristics, the *Bottleneck-Target* heuristic performs the best giving about 10-15% improvement in lifetime over our previous 1-hop Degree-Based heuristic and about 25-30% over LBP and DEEPS. *Sparse-OPT* and *Even-Target-Rate* also improve over the Degree-Based heuristic. We also ran simulations with a combination of the heuristics. $(2, 1)$ denotes the combination of heuristic 2, Bottleneck-Target, followed by heuristic 1, Sparse-OPT, and so on. For these experiments, the priority function was modified to implement the priority function of both heuristics in the same algorithm. For example, in $(2, 1)$, we first optimize the local bottleneck target (based on heuristic 2) and in case of ties, we break them by checking the degree to previously selected covers (based on heuristic 1). Other implementations follow similarly. The combination of heuristics give a better lifetime than the individual heuristics; the best hybrid was (2,3,1) plotted in Fig. 2.

In Fig. 3, we highlight the performance of the three heuristics as compared to LBP. The simulation is run for 40, 80 and 120 sensors, with 25 targets and a linear energy model. As can be seen, a similar trend in improvements is observed. Overall lifetime improvements are in the range of 25-30% over LBP as the baseline. Since DEEPS is a 2-hop algorithm, we also compared 2-hop versions of our proposed heuristics, where the target set $T(s)$ of each sensor is expanded to include $\cup_{s' \in N(s,1)} T(s')$ and the neighbor set is expanded to all 2-hop neighbors, i.e., $N(s, 2)$. The results are shown in Fig. 4. As can be seen, the 2-hop versions give a further gain in lifetime over DEEPS, with overall improvement of 35% for the hybrid of the three heuristics.

Finally, we highlight the comparison of the proposed heuristics against our previous work [12] in Fig. 5. We show results here for the median value of $n = 80$ sensors. The two series compare the 1-hop version of the Degree-Based heuristic against the 1-hop version of the OPT heuristics and then repeat these comparisons for the 2-hop versions of both. Overall, we achieve gains of 10-15% for the 1-hop heuristics and between 20-25% for the 2-hop heuristics. We also show (i) how far these heuristics are compared to the upper bound on network lifetime, and (2) that both linear and quadratic energy models follow similar trends.

6 Conclusion

In this paper, we address the problem of scheduling sensors to extend the lifetime of a Wireless Sensor Network. We examine the properties that an optimal schedule would exhibit and use these to design three new heuristics that work on the lifetime dependency graph model. Simulations show an improvement of 25-30% over the algorithms in [10, 11] and 10-15% over our previous work in [12] for the 1-hop algorithms. Two-hop versions show additional improvements over their counterparts. These heuristics are designed to work on higher level properties of the dependency graph. The net effect is a significant improvement in the state of art, with our algorithms performing no worse than 30-40% compared to the optimal network lifetime on an average. Future work includes dealing with the exponential nature of the problem space by designing heuristics that can effectively sample good quality local covers.

References

1. Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E.: A survey on sensor networks. IEEE Commun. Mag., 102–114 (2002)
2. Abrams, Z., Goel, A., Plotkin, S.: Set k-cover algorithms for energy efficient monitoring in wireless sensor networks. In: Third International Symposium on Information Processing in Sensor Networks, pp. 424–432 (2004)
3. Cardei, M., Du, D.Z.: Improving wireless sensor network lifetime through power aware organization. Wireless Networks 11(8), 333–340 (2005)
4. Slijepcevic, S., Potkonjak, M.: Power efficient organization of wireless sensor networks. In: IEEE International Conference on Communications (ICC), vol. 2, pp. 472–476 (2001)
5. Cardei, M., Thai, M., Li, Y., Wu, W.: Energy-efficient target coverage in wireless sensor networks. In: INFOCOM 2005, vol. 3 (March 2005)
6. Berman, P., Calinescu, G., Shah, C., Zelikovsky, A.: Power efficient monitoring management in sensor networks. In: Wireless Communications and Networking Conference (WCNC), vol. 4, pp. 2329–2334 (2004)
7. Miodrag, S.M.: Low power 0/1 coverage and scheduling techniques in sensor networks. UCLA Technical Reports 030001 (2003)
8. Dhawan, A., Vu, C.T., Zelikovsky, A., Li, Y., Prasad, S.K.: Maximum lifetime of sensor networks with adjustable sensing range. In: Proceedings of the International Workshop on Self-Assembling Wireless Networks (SAWN), pp. 285–289 (2006)
9. Lu, J., Suda, T.: Coverage-aware self-scheduling in sensor networks. In: 18th Annual Workshop on Computer Communications (CCW), pp. 117–123 (2003)
10. Berman, P., Calinescu, G., Shah, C., Zelikovsky, A.: Efficient energy management in sensor networks. In: Ad Hoc and Sensor Networks, Wireless Networks and Mobile Computing (2005)
11. Brinza, D., Zelikovsky, A.: Deeps: Deterministic energy-efficient protocol for sensor networks. In: Proceedings of the International Workshop on Self-Assembling Wireless Networks (SAWN), pp. 261–266 (2006)
12. Prasad, S.K., Dhawan, A.: Distributed algorithms for lifetime of wireless sensor networks based on dependencies among cover sets. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 381–392. Springer, Heidelberg (2007)
13. Dhawan, A., Prasad, S.K.: A distributed algorithmic framework for coverage problems in wireless sensor networks. In: Procs. Intl. Parallel and Dist. Processing Symp. Workshops (IPDPS), Workshop on Advances in Parallel and Distributed Computational Models (APDCM), pp. 1–8 (2008)

14. Tian, D., Georganas, N.D.: A coverage-preserving node scheduling scheme for large wireless sensor networks. In: WSNA: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications, pp. 32–41. ACM, New York (2002)
15. Zhang, H., Hou, J.: Maintaining sensing coverage and connectivity in large sensor networks. In: Ad Hoc and Sensor Wireless Networks, AHSWN (2005)
16. Xing, G., Wang, X., Zhang, Y., Lu, C., Pless, R., Gill, C.: Integrated coverage and connectivity configuration for energy conservation in sensor networks. ACM Trans. Sen. Netw. 1(1), 36–72 (2005)

In-Network Data Estimation for Sensor-Driven Scientific Applications[*]

Nanyan Jiang and Manish Parashar

Center for Autonomic Computing (CAC) &
The Applied Software Systems Laboratory (TASSL)
Department of Electrical and Computer Engineering
Rutgers University, Piscataway NJ 08855, USA
{nanyanj,parashar}@rutgers.edu

Abstract. Sensor networks employed by scientific applications often need to support localized collaboration of sensor nodes to perform in-network data processing. This includes new quantitative synthesis and hypothesis testing in near real time, as data streaming from distributed instruments, to transform raw data into high level domain-dependent information. This paper investigates in-network data processing mechanisms with dynamic data requirements in resource constrained heterogeneous sensor networks. Particularly, we explore how the temporal and spatial correlation of sensor measurements can be used to trade off between the complexity of coordination among sensor clusters and the savings that result from having fewer sensors involved in in-network processing, while maintaining an acceptable error threshold. Experimental results show that the proposed in-network mechanisms can facilitate the efficient usage of resources and satisfy data requirement in the presence of dynamics and uncertainty.

Keywords: Sensor system programming, In-network data estimation, Sensor-driven scientific applications.

1 Introduction

Technical advances in sensing technologies are rapidly leading to a revolution in the type and level of instrumentation of natural and engineered systems, and is resulting in pervasive computational ecosystems that integrate computational systems with these physical systems through sensors and actuators. This in turn is enabling a new paradigm for monitoring, understanding, and managing natural and engineered systems – one that is information/data-driven and that opportunistically combines computations and real-time information to model, manage, control, adapt, and optimize.

[*] The research presented in this paper is supported in part by National Science Foundation via grants numbers CNS 0723594, IIP 0758566, IIP 0733988, CNS 0305495, CNS 0426354, IIS 0430826 and ANI 0335244, and by Department of Energy via the grant number DE-FG02-06ER54857, and was conducted as part of the NSF Center for Autonomic Computing at Rutgers University.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 282–294, 2008.

Several scientific and engineering application domains, such as waste management [1], volcano monitoring [2], city-wide structural monitoring [3], and end-to-end soil monitoring system [4], are already experiencing this revolution in instrumentation. This instrumentation can also potentially support new paradigms for scientific investigations by enabling new levels of monitoring, understanding and near real-time control of these systems. However, enabling sensor-based dynamic data-driven applications presents several challenges, primarily due to the data volume and rates, the uncertainty in this data, and the need to characterize and manage this uncertainty. Furthermore, the required data needs to be assimilated and transported (often from remote sites over low bandwidth wide area networks) in near real-time so that it can be effectively integrated with computational models and analysis systems. As a result, data in most existing instrumented systems is used in a post-processing manner, where data acquisition is a separate offline process.

The overall goal of this research is to develop sensor system middleware and programming support that will enable distributed networks of sensors to function, not only as passive measurement devices, but as intelligent data processing instruments, capable of data quality assurance, statistical synthesis and hypotheses testing, as they stream data from the physical environment to the computational world [5]. This paper specifically investigates abstractions and mechanisms for in-network data processing that can effectively satisfy dynamic data requirements and quality of data and service constraints, as well as investigate tradeoffs between data quality, resource consumptions and performance. In this paper, we first present the *iZone* programming abstractions for implementing in-network data estimation mechanisms. We then explore optimizations that can use the spatial and temporal correlation in sensor measurements to reduce estimation costs and handle sensor dynamics, while bounding estimation errors. For example, an appropriate subset of sensors might be sufficient to satisfy the desired error bounds, while reducing the energy consumed. The optimized in-network data estimation mechanisms are evaluated using a simulator. The evaluations show that these mechanisms can enable more efficient usage of the constrained sensor resources while satisfying the applications requirements for data quality, in spite of sensor dynamics.

The rest of the paper is organized as follows. Section 2 describes the *iZone* programming abstraction and in-network data estimation mechanisms. Section 3 presents space, time and resource aware optimizations for in-network interpolation. Section 4 presents an experimental evaluation. Related work is discussed in Section 5. Finally, Section 6 presents conclusions.

2 In-Network Data Estimation

Scientific applications often require data measurements at pre-defined grid points, which are often different from the locations of the raw data provided directly by the sensor network. As a result, a sensor-driven scientific/engineering application requires a virtual layer, where the logical representation of the state of the environment provided to the applications may be different from the raw measurements obtained from the sensor network. The *iZone* abstractions described in this section enables applications to specify such a virtual layer as well as implement the models (e.g., regression models, interpolation functions, etc.) that should be used to estimate data on the virtual layer from sensor readings.

2.1 The *iZone* Abstraction

As mentioned above, there is often a mismatch between the discretization of the physical domain used by the application and the physical data measured by the sensor network and as a result, data from the sensors has to be processed before it can be coupled with simulations. The goal of the *iZone* abstraction is to support such an integration. It essentially abstracts away the details of the underlying measurement infrastructure and hides the irregularities in the sensor data by virtualizing the sensor field. The result is a consistent representation over time and space to match what is used by the simulations.

The *iZone* itself is thus a representation of the neighborhood that is used to compute a grid point of the virtual layer, and can be specified using a range of coordinates, functions, etc. The *iZone* abstraction enables the implementation of the estimation mechanisms within the sensor network. For example, interpolation algorithms, such as regressions, inverse distance weighing (IDW), and kriging, require the definition of an interpolation zone, an *iZone*, which defines the neighborhood around the grid point to be estimated, and such neighborhood is then used to compute that interpolation point. Note that for several interpolation algorithms, this zone may change on the fly based on the constraints provided by the application. The *iZone* abstraction provides operators for obtaining sensor measurements corresponding to the region of interest, as well as for defining in-network processing operators to compute a desired grid point from sensor values of this region. The semantics of operators of *discover*, *expand*, *shrink*, *get*, *put* and *aggregate* are listed in Table 1.

Once an *iZone* is defined, computing the data value at a grid point consists of (1) identifying a master node that coordinates the estimation process, which could be the sensor node that is closest to the grid point and has the required capabilities and resources, (2) discovering the sensors in the *iZone* that will be

Table 1. The *iZone* operators

Operator	Semantics
discover	Discover sensors within an *iZone*
expand	Expand an *iZone* by adding additional sensors
shrink	Shrink an *iZone* by removing sensors
get	Collect data from sensor(s) in the *iZone*
put	Send data to sensor(s) in the *iZone*
aggregate	Aggregate sensor data using reduction operators, such as max, min, weighted_avg, etc.

used in the estimation, (3) planning the in-network estimation strategy based on desired cost/accuracy/energy tradeoffs, and (4) performing the estimation and returning the computed data value at the desired grid point.

3 STaR: Space, Time and Resource Aware Optimization

This section explores temporal and spatial correlations in the sensor measurements to reduce costs and handle sensor dynamics, while bounding estimation errors for a given *iZone*. For example, a subset of the sensors in an *iZone* may be sufficient to satisfy the desired error bounds while reducing the energy consumed. Further, temporal regression models can be used to handle transient data unavailability, which may otherwise lead to a significant increase in costs and energy consumption (see Section 4.1).

3.1 Saving Energy Using *iSets*

Typically, for a densely deployed sensor network, a subset of sensors consisting of members in an *iZone* may be sufficient to meet the quality requirements for in-network data estimation. In this case, sensors in an *iZone* are divided into multiple interpolation sets, i.e., *iSets*, each of which can be used to estimate the data points while still satisfying the error bounds, and reducing costs and energy consumed. The *iSets* are generated and maintained at runtime to balance cost and energy as well as to tolerate failures.

The problem of generating the *iSets* can be formalized as follows: assume that an *iZone* Z is divided into m exclusive subsets $\{S_1, S_2, ..., S_m\}$ (such that $S_i \cap S_j = \emptyset$ and $S_1 \cup S_2, ..., \cup S_m = Z$), and each subset k $(k = 1, 2, ...m)$ contains N_{a_k} number of sensors. The objective is to find "best" collection of *iSets* that satisfies data quality requirements.

The *iSets* should satisfy three requirements: (1) the interpolation error for each *iSet* should be less than the specified error tolerance; (2) the average number of sensor measurements for each *iSet* should be minimized in order to reduce the energy consumed; (3) the average aggregated error (i.e., $1/m \sum_k err(S_k)$) should be minimized in order to achieve data quality whenever possible. In addition to these basic requirements, further constraints may be added to satisfy additional resource consumption and scheduling requirements. For example, the sizes of the *iSets* should be similar to make resource consumption more balanced and the scheduling easier. Similarly, the variance of interpolation errors across the *iSets* should be as small as possible.

The *iZone* is thus divided into m *iSets*, only one *iSet* of which needs to be active at a time. These *iSets* can now be scheduled in a round-robin fashion. Note that, as the number of *iSets* increases (i.e., the average size of *iSets* decreases), the efficiency of the approach increases as well. The generation and maintenance of *iSets* is illustrated in Figure 1 and is described below.

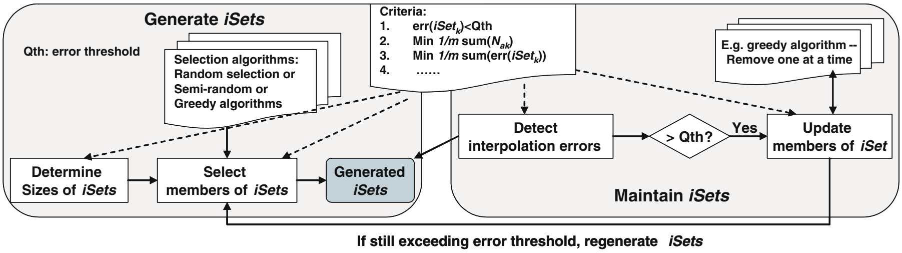


Fig. 1. An overview of generating and maintaining *iSets*

3.2 Generating *iSets*

Determining the sizes of *iSets*: The appropriate size of the *iSets* used for interpolation is determined based on the specifications provided by the application, and computed (offline or online) using a stochastic approach as follows. The size of an *iSet* is first initialized to 3 (i.e., $k = 3$). The sensors used for interpolation tests are randomly selected, and the number of tests is set to some reasonable number (i.e., $N_{tr} = 50$). If the interpolation error is within the error tolerance threshold Q_{th}, the successful interpolation counter *succ* is incremented. If the success rate (i.e., $succ/N_{tr}$) is greater than a specified percentage θ, the algorithm terminates and the current size of the *iSet* is returned as the desired *iSet* size. If the success rate is less than θ after all the tests are completed, the size of *iSet* is incremented and the procedure is repeated until the size equals the size of the *iZone*. Note that this is only done once.

Selecting members of the *iSets*: Once the size of an *iSet* is determined, the members of each *iSet* are selected so as to satisfy the criteria discussed earlier. A straightforward approach is to use an exhaustive search to find the optimal collection of *iSets*. This approach is obviously expensive for reasonably sized *iZones*, and as a result, we propose approximate algorithms, i.e., the random, semi-random and greedy algorithms, for finding near optimal *iSets*, as described below.

Random algorithm. Given N sensors in an *iZone*, this algorithm randomly generates m mutually exclusive *iSets*, each approximately of size k (i.e., $N_{a_k} \approx k$), and $\sum_{k=1}^{m} N_{a_k} = N$. The interpolation error of each *iSet* is then evaluated. If the error for every *iSet* is below the error threshold, Q_{th}, the aggregated error across all the *iSets* is computed and saved. This process is repeated several times. The collection of *iSets* that leads to the smallest aggregated error is finally selected as the initial collection of *iSets*. The number of trials N_{tr} may be explicitly specified or computed based on observed (or historical) data. The actual value depends on the characteristics of sensor data. For example, for the dataset used in the experiments in the paper (see Section 4), a suitable value is between 50 and 100.

Semi-random algorithm. This algorithm is based on the heuristic that if the sensors in an *iSet* are more uniformly distributed across the *iZone*, the estimation has higher chance to be more accurate. This algorithm attempts to assign neighboring nodes to different *iSets*. This is done using locality preserved space filling curves [6] as the indexing mechanism. First, each sensor is indexed using the Hilbert space filling curve (SFC) based on their locations. Within a given *iZone*, sensors with neighboring identifiers are then virtually grouped based on their SFC indices, so that the size of each virtual group is equal to the number of *iSets* required. For example, if m *iSets* are needed, each group would have m members. The *iSets* are now constructed by randomly selecting one sensors from each of the virtual groups.

Greedy algorithms. Three of greedy algorithms are also devised to select appropriate sensors for the *iSets*. These algorithms are described below.

Greedy Algorithm 1 – "Remove one node at a time": In this algorithm, we start with one *iSet* containing all sensors in the *iZone*. Sensors are then removed one at a time until the desired *iSet* size is achieved. The removed sensor is the one that leads to minimal interpolation error at each step. That is, given the *iSet* k, sensor node j is removed from the *iSet*, such that

$$j = \arg \min_{i \in S_k} err(S_k - i).$$

To find the optimal *iSets*, the last two *iSets* are chosen using one of the other methods, such as random algorithm or the second greedy algorithm.

Greedy Algorithm 2 – "Add one node at a time": The second algorithm starts with a single sensor node in the initial *iSet*, and adds one node at a time while maintaining the interpolation error constraints. That is, given the *iSet* k, node j is added to the *iSet* to minimize the interpolation error, such that

$$j = \arg \min_{i \in Z \setminus S_k} err(S_k \cup i).$$

This process is repeated until all nodes are assigned to *iSets*.

Greedy Algorithm 3: This algorithm uses the heuristic that nodes which are far from each other are less correlated and as a result are good candidates to add to an existing *iSets*. The idea is to select sensors that are far from the last selected sensor. To implement this algorithm, we use the SFC-based indexing method described as part of the semi-random algorithm. Sensor nodes are first indexed using the Hilbert SFC. Virtual groups with sizes equal to number of *iSets* are then formed based on their SFC indices.

The algorithm is initialized by randomly assigning sensors of one virtual group to each *iSet*. Next, sensors are permutated from one of remaining virtual groups, and are mapped to each of m *iSets*. The permutation leading to the least aggregated interpolation error is added to each of them respectively at a time. This step is repeated until all the sensors are assigned to the *iSets*. Since the order in which virtual group are processed has the impact on the interpolation errors, a pre-processing step is used to find the sequence(s) to be used. Note that this only needs to be done once.

Next, we describe how to maintain *iSets* when the underlying system changes at runtime.

3.3 Maintaining *iSets* at Runtime

Due to the dynamics of underlying physical environment and the sensor system, currently valid *iSets* may not satisfy data quality requirements in the future. As a result, mechanisms are needed to maintain *iSets* to ensure that they continue to meet data quality requirements. In this section, we describe such mechanisms. In this discussion, we assume that the sensor network is clustered to construct a two level self-organizing overlay of sensors, in which cluster heads perform coordination of the *iZone*.

Our overall approach is as follows. First, interpolation errors are tracked by using localized error models at each individual cluster. The error models are localized so that a violation of error thresholds can be detected locally without communicating with other clusters. When a threshold violation is detected, a greedy algorithm is used to update the involved *iSet* to improve estimation quality whenever possible.

Generating models for interpolation errors: It is noted that interpolation errors are often correlated with relevant sensor measurements. As a result, regression models can be used to describe the relationship, $e(t) = f(v_k(t))$, between interpolation errors $e(t)$ and the current measurement $v_k(t)$ of sensor k. A combination of offline and online estimation methods can be used to learn such a relationship, in which the coarse trends of error models are learned using offline methods using historical data, while specific local parameters can be learned at runtime. For example, an offline study may suggest that a regression model of degree one, i.e., $a_1 v_k + b_1$, should be used. The model parameters a_1 and b_1 are estimated using previous values of sensor k and the corresponding interpolation errors at runtime at individual cluster heads. These models can then be used to estimate interpolation errors using measurement of sensor k from local cluster.

Maintaining *iSets* using a greedy algorithm: When an *iSet* only temporarily exceeds error thresholds, the greedy "remove one at a time" algorithm can be used at each cluster to temporarily remove sensor measurements from that *iSet*. Each cluster makes recommendation of which node(s) to remove, and the recommendation that provides the least interpolation error is enforced. Note that if the interpolation error still exceeds error threshold, the *iSets* needs to be regenerated.

Transient unavailability using temporal estimation: Since temporarily unavailability of scheduled sensors requires re-collection of raw data and thus results in expensive communication and energy consumption, temporal models are used to estimate the missing sensor measurement. The idea is to use the fact that neighboring sensor nodes would experience similar changes. As a result, samples from neighboring sensors can be used to facilitate the estimation of temporal model parameters, such as degree of regression model, length of time-series. Note that these parameters would change as the underlying physical characteristics vary. The actual coefficients of temporal model are determined based on previous values of the missing sensor data. The evaluations of these optimized in-network mechanisms are presented next in Section 4.

4 Experimental Evaluation

In this section a simulator is used to evaluate the performance of in-network data estimation mechanisms. The simulator implements the space, time, and resource aware optimization mechanisms for realistic scenarios. The scenarios in the experiments are driven by a real-world sensor dataset obtained from an instrumented oil field with 500 to 2000 sensors, and consisting of pressure measurements sampled 96 time per day. A two-tiered overlay with about 80 clusters is initialized. More powerful nodes are elected as cluster heads and also perform the in-network estimations.

In each experiment, about 500 instances of in-network interpolations are performed on pressure values obtained from simulated sensor nodes. Communication costs are evaluated with and without optimizations. The accuracy and costs are also evaluated in the presence of dynamics of physical environments and sensor systems. Finally, the cost of generating *iSets* is examined using the random, semi-random and greedy algorithms. The primary metrics used in the evaluation are communication cost, measured in terms of number of messages transmitted within the network, and accuracy, measured in terms of relative or absolute interpolation errors.

4.1 Communication Costs

The current *iZone* prototype implements a distributed in-network mechanism, in which parameters corresponding to the estimation model are first computed at the cluster heads. The cluster heads coordinate the estimation process, and distribute those parameters to the selected *iZone* sensors. A decentralized energy-efficient aggregation scheme is then used to estimate the data. To simulate transient unavailability of sensors, each sensors are given the same unavailability rates, which are the frequencies that scheduled sensors are not available at the time of interpolation. The communication costs are normalized to the cost of the baseline centralized approach, where a coordinator sensor collects raw measurement from selected *iZone* sensors and does the estimation. As plotted in Figure 2, the distributed approach performs best when the sensor system is static. With a small unavailability rate of 0.5%, the communication cost increases by over 50% for an *iZone* radius of 60, and over 4 times for a radius of 140. The cost also increases as the unavailability rate increases.

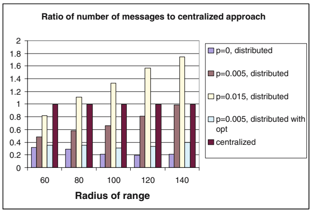


Fig. 2. Communication cost in the presence of sensor dynamics

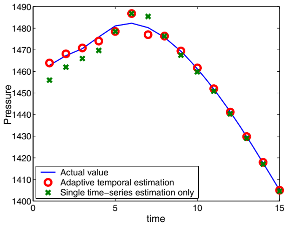


Fig. 3. Adaptive temporal interpolation

Effectiveness of temporal estimation: In this experiment, the performance of using temporal estimation for temporarily unavailable sensor data is evaluated. Model parameters such as the order of regression functions, and the window size of historical data used for the estimation, are chosen at runtime. The estimation parameters varies over time. For example, the window size of historical data is changed from 8 to 6 after the sixth measurement, and the degree of regression models is changed from 3 to 2 after the fifth measurement. The adaptive temporal estimation using spatial-temporal information from neighboring nodes (i.e., circles in Figure 3) is much closer to the actual values (i.e., solid line) than that using only historical measurements from sensors with temporarily unavailable data (i.e., crosses in Figure 3). This is because when using the spatial-temporal models of neighboring nodes, the changes of the underlying physical characteristics can be better detected than that using the historical data only from temporarily unavailable sensor. Furthermore, as plotted in Figure 2, by using adaptive temporal estimations for transient unavailability, the communication cost is reduced by about 25% for a radius of 60, and about 60% for a radius of 140.

4.2 Costs of Generating *iSets*

In these experiments, the costs and effectiveness of using random, semi-random and greedy algorithms to generate *iSets* for in-network interpolation tasks are investigated, and three *iSets* are formed within the given *iZones* for this set of experiments.

Costs of random-based algorithms. The histograms of average interpolation errors for each of three generated *iSets* are plotted in Figure 4 using random and semi-random algorithms respectively. The probability of smaller average interpolation errors using the semi-random algorithm is much higher than that using the random algorithm. For example, for an average interpolation error less than 0.2%, the semi-random algorithms (e.g., with probability about 32%) have higher probability to generate *iSets* meeting quality requirements than that using the random algorithms (e.g., with probability about 12%).

In Figure 5, the standard deviation of interpolation errors is examined for the two algorithms. The random algorithm gives much larger variation than the semi-random algorithm. For the random-based algorithms, within the same range of interpolation error (i.e., 0.62%), the standard deviation of random method is still much larger than that of semi-random algorithm. This tells us that the semi-random algorithm is generally more effective in finding the collections of *iSets* having both small interpolation errors and a small variation of such errors.

In Figure 6, the number of collections of *iSets* is counted in terms of the absolute average interpolation errors and deviations (i.e., δ) among *iSets*. A larger number of candidates that meet these requirements is available when the semi-random algorithm is used than when the random algorithm is used. For example, with small variance 1 (i.e., $\delta = 1$), the available number of collections using semi-random algorithm is five more times than that using random

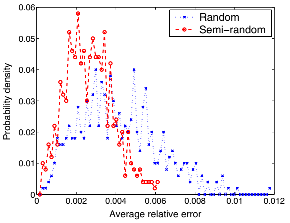


Fig. 4. Histograms of interpolation errors of generated *iSets*

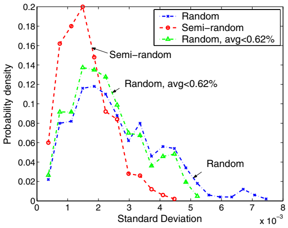


Fig. 5. Histograms of variation of interpolation errors of generated *iSets*

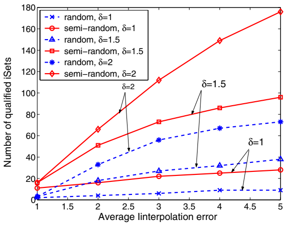


Fig. 6. Effectiveness of random-based algorithm

algorithm, which indicates the effectiveness using the semi-random algorithm especially with higher quality requirements.

Effectiveness of greedy algorithms. First the three greedy algorithms, as well as random and semi-random algorithms are compared in terms of interpolation errors. Three *iSets* are generated in this example. As shown in Figure 7, the first greedy algorithm, i.e., "remove one at a time", behaves well for most of the generated *iSets* except the last one. For this algorithm, the last *iSet* exhibits high error since it has no chance to exploit local optimization. The last two *iSets* are thus chosen using other algorithms, such as random algorithms or "add one at a time" greedy algorithm. The second algorithm, "add one at a time" gives relatively balanced results. The overall errors are higher than those of random and semi-random algorithms. The third greedy algorithm gives good accuracy performance comparing to random and semi-random algorithms. The tradeoff is that it needs pre-processing to find good sequence of the next explored sensors. However, the pre-processing could be performed offline and the costs are much less than that for random and semi-random algorithms.

4.3 Tradeoffs between Accuracy and Energy Consumption

In this section, the tradeoff of accuracy and energy consumption is examined. The energy consumption is normalized to one when all sensors in an *iZone* are active

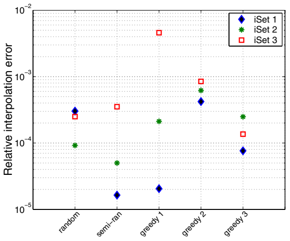


Fig. 7. Interpolation errors of algorithms generating *iSets*

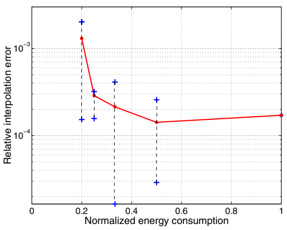


Fig. 8. Tradeoff between interpolation error vs. energy consumption

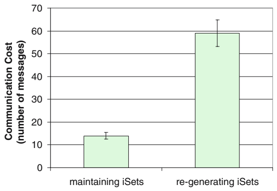


Fig. 9. Cost of maintaining *iSets*

at each time. As shown in Figure 8, as the sizes of *iSets* becomes smaller, less energy is consumed and the interpolation errors become greater. For example, when half of the nodes are active, approximately half of the energy can be saved. The maximum error is slightly greater than when using all nodes. When only one third or one fourth of nodes are active, the maximum error is similar to that of using only half of all nodes. This means that only a portion of active sensors is able to meet accuracy requirements while saving additional energy. However, when only one fifth of nodes are active, the errors (both maximum and average) become much larger and cannot meet application requirements anymore.

4.4 Cost of Maintaining *iSets*

In this experiment, the communication cost of maintaining *iSets* at local cluster heads is examined. The cost primarily consists of exchanging information, such as calibration information, identifiers of temporarily removed sensors between cluster heads. As shown in Figure 9, the cost of maintaining *iSets* at cluster head introduces much less communication overhead than regenerating *iSets*. Furthermore, the original error before removing a sensor is .06069% (threshold is .06%), and after removing one of them using greedy algorithm, the result becomes .00922%, which is far lower than the threshold. This method is quite effective when the change of physical phenomenon is just temporary. After this, the original *iSet* can be used again with the error rate lowered to .04681%. As a result, this method significantly reduced the frequency to regenerate the whole *iSets* in the *iZone*.

5 Related Work

There are some recent research efforts [7,8] that use the concept of virtual sensors to support sensor applications. VNLayer [7] provides abstraction layers that mask uncertainty of underlying sensor networks through consistency management. Virtual sensor [8] provides a virtual sensor model and application APIs to support heterogeneous aggregation and hierarchical specifications. However, the underlying concepts and implementation of the system described in this paper is quite different from these approaches in that it uses virtualization to address the mismatch between the instrumentation of the physical domain and its discretization in the computational model, rather than to create, for example, a virtual sensor for a derived data type.

Data aggregation is an essential functionality in sensor networks, and has been addressed by a number of research efforts [9,10]. Optimizations techniques such as aggregation trees are used to resolve queries efficiently. Homogeneous aggregation operations are supported. The approach presented in this paper supports used-defined functions using in-network coordination and optimization mechanisms. The sensor selection schemes are also closely related to our work. The sensor selection approach described in [11] uses approximation algorithms to select near-optimal subsets of k sensors that minimize the worst-case prediction error. Entropy-based approaches [12] are used for the sensor selection problems of target tracking and localization applications. The goal of our proposed algorithms is to find a *"best" collection* of subsets, *all* of which satisfy the error tolerance rate (and minimize aggregated errors), while saving and balancing the energy consumptions among sensors in the long run.

6 Conclusion

This paper investigated abstractions and mechanisms for in-network data processing that can effectively satisfy dynamic data requirements and quality of data and service constraints, as well as investigate tradeoffs between data quality, resource consumptions and performance. Specifically, the proposed mechanisms (i) allow flexibility in the specification of relevant subsets of a sensor network with *iZones* and *iSets*; (ii) explore space, time and resource aware optimizations that utilize the spatial and temporal correlation among sensor measurements to reduce costs while bounding estimations errors; (iii) are robust with respect to network dynamics; and (iv) provide a virtualization of the physical sensor grid to match the representation of the physical domain used by the models, and can dynamically discover and access sensor data independent of any change to the sensor network itself. Experimental results show that the proposed in-network mechanisms can facilitate the efficient usage of constraint resources and satisfy data requirement in the presence of dynamics and uncertainty.

References

1. Parashar, M., Matossian, V., Klie, H., Thomas, S.G., Wheeler, M.F., Kurc, T., Saltz, J., Versteeg, R.: Towards dynamic data-driven management of the ruby golch waste repository. In: Proceedings of the Workshop on Distributed Data Driven Applications and Systems, International Conference on Computational Science, ICCS (2006)
2. Werner-Allen, G., Lorincz, K., Ruiz, M., Marcillo, O., Johnson, J., Lees, J., Welsh, M.: Monitoring volcanic eruptions with a wireless sensor network. In: Second European Workshop on Wireless Sensor Networks (2005)
3. Kottapalli, V.A., Kiremidjiana, A.S., Lyncha, J.P., Carryerb, E., Kennyb, T.W., Lawa, K.H., Lei, Y.: Two-tiered wireless sensor network architecture for structural health monitoring. In: SPIE's 10th Annual International Symposium on Smart Structures and Materials (2003)
4. Szlavecz, K., Terzis, A., Musaloiu-E., R., Cogan, J., Small, S., Ozer, S., Burns, R., Gray, J., Szalay, A.S.: Life under your feet: An end-to-end soil ecology sensor network, database, web server, and analysis service. MSR-TR-2006-90 (2006)
5. Jiang, N., Parashar, M.: Programming support for sensor-based scientific applications. In: Proceedings of the Next Generation Software (NGS) Workshop in conjunction with the 22nd IEEE International Parallel and Distributed Processing Symposium, IPDPS (2008)
6. Sagan, H.: Space-Filling Curve. Springer, Heidelberg (1995)

7. Brown, M., Gilbert, S., Lynch, N., Newport, C., Nolte, T., Spindel, M.: The virtual node layer: A programming abstraction for wireless sensor networks. ACM SIGBED Review 4(3), 7–12 (2007)
8. Kabadayi, S., Pridgen, A., Julien, C.: Virtual sensors: abstracting data from physical sensors. In: Proceedings of the 2006 International Symposium on World of Wireless, Mobile and Multimedia Networks (WoWMoM), pp. 587–592 (2006)
9. Yao, Y., Gehrke, J.E.: The cougar approach to in-network query processing in sensor networks. Sigmod Record 31(3) (2002)
10. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TAG: a Tiny AGgregation service for Ad-Hoc sensor networks. In: Proceedins of the USENIX Symposium on Operating Systems Design and Implementation (2002)
11. Das, A., Kempe, D.: Sensor selection for minimizing worst-case prediction error. In: International Conference on Information Processing in Sensor Networks, IPSN 2008 (2008)
12. Zhao, F., Shin, J., Reich, J.: Information-driven dynamic sensor collaboration for tracking applications. IEEE Signal Processing Magazine (2002)

Localization in Ad Hoc and Sensor Wireless Networks with Bounded Errors

Mark Terwilliger[1,2], Collette Coullard[2], and Ajay Gupta[1]

[1] Computer Science Department, Western Michigan University, Kalamazoo, MI, USA
[2] Computer Science Department, Lake Superior State University, Sault Ste. Marie, MI, USA

Abstract. With the proliferation of wireless networks and mobile computing devices, providing location-aware technology and services to new applications has become important for developers. Our main contribution is an efficient location discovery algorithm that bounds the localization error. Providing an efficient localization technique is critical in resource-constrained environments that include mobile devices and wireless networked sensors. Applicable in centralized and distributed environments, our algorithm, based on finding the smallest circle enclosing the intersection of n disks, runs in $O(n^2)$ time. We then extend our work to the problem of finding the smallest disk that includes the set of points common to n disks and excluded from the interiors of m other disks. Finally, we show performance results from the implementation of our algorithms in which, under some conditions, localization estimates for 500 nodes in a 500x500 ft region can be found with a mean error of one foot and a two-foot error bound.

Keywords: Localization, error bound, sensor network, position, distance estimates.

1 Introduction

Advancements in low-power electronic devices integrated with wireless communication capabilities and sensors have opened up an exciting new field in computer science. Wireless sensor networks (WSN) can be developed at a relatively low-cost and can be deployed in a variety of different settings. A WSN is typically formed by deploying many sensor nodes in an ad hoc manner. These nodes sense physical characteristics of the world. The sensors could be measuring a variety of properties, including temperature, acoustics, light, and pollution. Base stations are responsible for sending queries to and collecting data from the sensor nodes. Some of the main characteristics of a networked sensor include: (1) small physical size, (2) low power consumption, (3) limited processing power, (4) short-range communications, and (5) a small amount of storage.

Localization is the process of determining the positions of nodes in an ad hoc network. It is an important problem that has attracted much attention in this decade [1]. Providing robust localization services remains a fundamental research challenge facing the entire WSN development community [2].

A common practice when locating an object is to use estimated distances to known positions, or *anchors*. Suppose two objects are actually separated by distance d . If the

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 295–308, 2008.

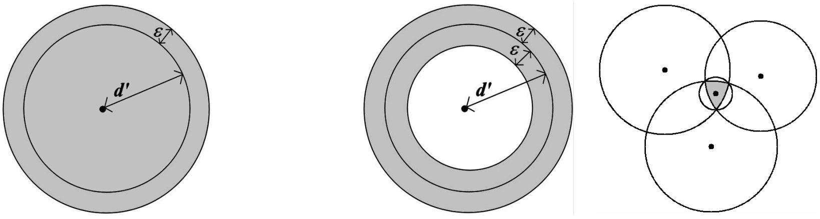


Fig. 1. (i) A disk, (ii) A washer, (iii) smallest enclosing circle covering intersection of disks

estimate to a position is given by d' and the possible error for this distance is $\pm\varepsilon$, then the object must be located within a disk with radius $d'+\varepsilon$ as shown in Fig. 1(i). If we remove the inside disk with radius $d'-\varepsilon$, we can further say that the object must be located in a washer as illustrated in Fig. 1(ii).

If we have distance bounds $d'+\varepsilon$ from *n* known positions to an object, we know that the object must be located in the intersection of those n disks. We find the smallest circle enclosing this intersection. By choosing the center of this circle as the location estimate, we guarantee that the localization error cannot be larger than the circle's radius. See Fig. 1(iii).

In this paper, we provide algorithms that find the smallest enclosing circle for both the intersection of disks, as well as washers. In addition to providing each step of our location discovery algorithms and we analyze their efficiency. We first provide an $O(n^3)$-time complexity algorithm for finding a location estimate along with a corresponding error bound. We then use a novel approach to improve our technique to $O(n^2)$ for the intersection of n disks. Our technique has the following advantages.

1) For every location estimate that is made, a bound on the maximum possible error is also provided.

2) The network *just* has to be connected. As opposed to many localization techniques, we have no restrictions on the number of neighbors each node must have to other nodes or anchors.

3) The algorithms work in both a centralized, as well as, a distributed environment.

4) This technique applies to both ad hoc mobile computing and sensor networks.

5) The algorithms are independent of the ranging technique used by nodes to estimate distances between neighbors.

The remainder of this paper is organized as follows: In Section 2, we define specifically the problems that we are attempting to solve. We describe related work in Section 3. We provide our solution to the Disk Problem in Section 4 and the Washer Problem in Section 5. Performance results are given in Section 6. We make final conclusions and talk about future work in Section 7.

2 Problem Description

The standard localization problem can be defined as follows: "Reconstruct the positions of all the nodes in a network given the relative pairwise distances among all the nodes that are within some radius r of each other." While we are given 1-dimensional measures of the relative distances, we are required to compute the positions either in a 2-dimensional or a 3-dimensional space, which makes the problem interesting and challenging. Throughout this paper, without loss in generality, we target our algorithms for the resource constrained and energy-critical WSNs, however, our solutions are applicable to more general wireless ad-hoc networks.

The localization problem is even more important in wireless sensor networks for the following reasons:

1. Many WSN protocols and applications simply assume that all nodes in the system are location-aware.
2. If a sensor is reporting a critical event or data, we must know the location of that sensor.
3. If a WSN is using a geographical routing technique, all of the nodes must be aware of their location.

Given known exact distances between neighbors, the localization problem has been shown to be NP-hard [3]. An added challenge is the fact that in practice, the exact distances between pairs of sensor nodes are not known. Instead, estimates are used to approximate the distances.

Therefore, there are two sources of errors in localization techniques – errors in relative pairwise distance estimates and even if exact distances are known, errors in computing the global coordinates. One must try to minimize these for heuristic techniques to be effective.

This paper addresses the problem of finding the location of *all* the nodes in a network given the location of a small subset of the nodes and estimates of the relative distances between pairs of nodes (i.e., relative distances are not known exactly). Our solution is in the form of a point estimate along with an error bound.

The first problem that we address is called the **Disk Problem**: Given centers and radii of n distinct disks, the goal is to find the center and radius of the smallest disk $D^* = [x^*, y^*, r^*]$ that includes the intersection of the n input disks.

The second problem that we address is the **Washer Problem**. This is similar to the Disk Problem, but the goal is to now find the center and radius of the smallest disk $D^* = [x^*, y^*, r^*]$ that includes the set of points included in all of the input washers.

3 Related Work

Most localization techniques consist of two steps or phases. In the first phase, distances or angles are measured between known points and the object to be located. This first phase is referred to as the **ranging phase**. In the second phase, these distance or angle measurements are combined to produce the location of the object. This phase is referred as the **localization phase**.

Some of the prominent techniques for the ranging phase include: 1. Received Signal Strength Indicator (RSSI), 2. Incremental Stepping of Transmission Power, 3. Time of Arrival (ToA), and 4. Angle of Arrival (AoA) [1,4].

Depending on the method used for ranging, an appropriate localization technique is applied in the second phase. The following localization strategies have been proposed [1,4]:

1. **Trilateration** – This is one of the more popular strategies and is used when the exact distances between known points and an object to be located are available. When the distance between an object and three points are given, the object's location can be computed as the intersection of three circles.
2. **Bounded Intersection –** The trilateration technique works well when the three circles intersect at a single point, but this is rarely the case when estimates are used in ranging. When using incremental stepping of transmission power for ranging, maximum values can be used for estimating the distances. The object to be located would fall into a geometric region that is the intersection of three circles.
3. **Triangulation –** The triangulation method is useful if the angle between two objects can be measured.
4. **Maximum Likelihood –** When estimates are used for ranging, it is possible that region of intersection is empty. This will occur if at least one ranging estimate is too small. One method that overcomes this problem selects the point for localization that gives the minimum total error between measured estimates and distances.

The most obvious solution to the localization problem is to simply equip every node with its own GPS device. This strategy might be feasible in some scenarios, but it suffers from several limitations of GPS such as it does not work indoors or when the line-of-sight is blocked. The size, cost and power consumption of a GPS receiver are also factors that make it impractical to equip all of the nodes in a WSN with this technology. Therefore, one must develop alternate low-cost and low-power solutions.

We presented one such solution based on evolution strategies that combined information about anchor positions with distance estimates between neighboring nodes [5]. In this paper, we present a deterministic algorithm that includes an error bound with each localization estimate.

The current landscape of location sensing systems is filled with a variety of technologies. The most popular system, GPS [6], uses radio time-of-flight lateration via satellites, but has some limitations. A good discussion of location systems is found in [1]. Most of the location systems discussed rely on known positions or distances in the location or calibration process and they rely on an a priori *infrastructure*. This leads to two problems: (1) The system will not scale well to a large topology, and (2) It is very difficult to do location sensing in an ad-hoc manner.

The problem of finding the location of *all* nodes in a wireless sensor network given the location of a subset of nodes has been approached by many researchers. A system called AHLoS (Ad-Hoc Localization System) [7] assumed that *beacon* nodes are aware of their positions. The rest of the nodes in the system are referred to as *unknown*, as these nodes will try to discover their location. The beacon nodes broadcast their location. An unknown node within range of three or more beacons estimates its position to minimize the mean square error. A technique called *iterative multilateration* is then used to handle the localization of all the nodes in the system. The accuracy of ranging in AHLoS was very precise, but it comes with a substantial cost in CPU power, energy consumption, and hardware circuitry. The percentage of beacons necessary to perform collaborative multilateration is relatively high. For example, for 90% of the network to localize in a network of 300 nodes, it is necessary for 45 of these nodes to be designated as beacons.

Many of the other existing localization algorithms, such as ABC [8], TERRAIN [9], and the work proposed by Meguerdichian et al [10], consist of two phases: 1) Estimate Position, and 2) Iterative Refinement The iterative refinement phase consists of approximately 25 iterations of **every** node sending its location to all of its neighbors. This process must be repeated when changes to the topology occur. Although this technique seems to provide good results as far as localization accuracy is concerned, the energy utilization in the wake of every node continuously broadcasting its location can be overwhelming, particularly when energy is one of the most precious resources for nodes in sensornets.

Our algorithms do not include an iterative refinement phase, making them more efficient as far as energy conservation is concerned. In [11], rectangular bounds were placed around possible positions of nodes by using linear programming. To the best of our knowledge, no one has done work on bounding the localization error by finding the intersection of disks or washers.

4 The Disk Problem

As described in the Introduction, our work relies on finding the smallest circle that encloses the intersection of an arbitrary number of input disks. The intersection of the disks is the region in which the object of interest lies. The center of the smallest enclosing circle is our estimate of its location; the radius is our bound on the error. By choosing the center as our estimate, it minimizes the error bound. In this section, we first present an $O(n^3)$-time algorithm for finding this smallest circle, analyze its computational complexity, and then describe how to improve it to $O(n^2)$. In [12], we prove the correctness of our algorithms.

$O(n^3)$ Algorithm for the Disk Problem

Input: Centers and radii of n distinct disks: $D_i = [x_i, y_i, r_i],\ i = 1,\ldots,n$,where $r_1 \le r_2 \le \cdots \le r_n$.

Output: Center and radius of the smallest disk $D^* = [x^*, y^*, r^*]$ that includes the intersection S_n of the n input disks.

1. [$O(n^2)$] Find all pairwise *intersection points* of the associated circles of the input disks. (There can be at most 2 intersection points for each pair of circles, for a total of $n(n-1)$ intersection points.)

2. [$O(n^3)$] For each intersection point, determine if this point lies in all of the input disks. Let $\{(a_k, b_k)\}$ be the set of these *corner points*, each of which lies in all of the input disks. (Although this step takes $O(n^3)$-time, we show that the number of corner points can be reduced to $O(n)$, and they can be found in $O(n^2)$ time.)

3. [$O(n^2)$] For each corner point that lies on the smallest input circle C_1, check to see if its antipodal point on C_1 lies in all of the input disks. If no corner points lie on C_1, then pick any antipodal pair of C_1 and check if both points lie in all of the input disks. If any antipodal pair lies in all of the input disks, return C_1 as the smallest enclosing circle. Otherwise, proceed to Step 4.

4. [$O(n)$] Return Smallest($\{(a_k, b_k)\}$), the smallest disk containing the $O(n)$ corner points $\{(a_k, b_k)\}$. Megiddo's linear programming algorithm finds the smallest disk containing a set of input points in linear time [13].

Next, we explain the steps of the algorithm, improve Step 2 to $O(n^2)$, and verify the $O(n^2)$ time complexity of the overall algorithm.

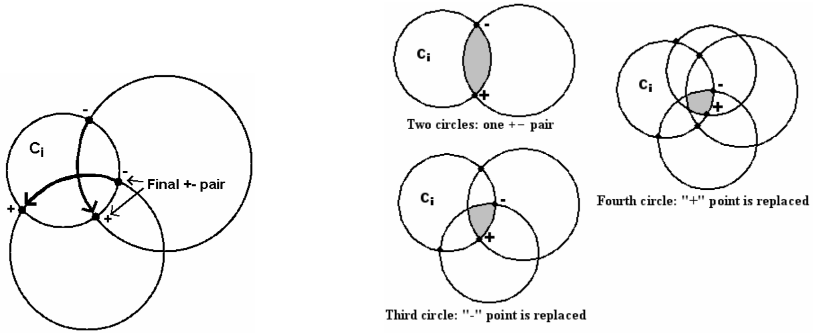
STEP 1: Finding the Pairwise Intersection Points

In [14], an $O(1)$ -time complexity algorithm is provided for finding the intersection of two circles. Using this algorithm $n(n-1)/2$ times, once for each pair of the n circles, we can find all pairwise intersection points in $O(n^2)$ -time.

STEP 2: Finding the Corner Points in $O(n^2)$ time

In [12], we establish that the number of corner points can be reduced to at most $2n-2$. Now, we will show how these corner points can be found in $O(n^2)$ time.

For each input circle C_i, order the intersection points with all circles $C_j : 1 \le j < i$, in a counter-clockwise (increasing-angle) direction. For each intersection point, label that point "+" if the segment from that point in the positive direction lies in both circles meeting at that point, and label it "$-$" otherwise. Traverse the circle in a positive-angle direction until a "+" is followed immediately by a "$-$". Those two points, which we will call a $+-$ *pair*, are the only corner points of S_j on C_i. This process is illustrated for three circles in Fig. 2(i). If there are two such $+-$ pairs, then we can conclude C_i contributes no corner points. Doing this for all input circles except for the smallest one, C_1, results in an $O(n^2 \log n)$ method for finding all the corner points, which is better than $O(n^3)$, at least.



Two circles: one +− pair

Fourth circle: "+" point is replaced

Third circle: "-" point is replaced

Fig. 2. (i) final plus/minus pair, (ii) processing plus/minus pairs

In this approach, we are finding all of the corner points of S_2, S_3, …, and S_n, and we proved this is at most $2n-2$ total points. A final step to eliminate those not in S_n, then can be done in $O(n^2)$ time.

We can modify the above approach to avoid the $O(n\log n)$ work required to sort each circle's intersection points, as follows. While processing circle C_i, keep the current $+-$ pair. If the "+" point of the next intersection pair is greater than the current "+" point, then replace the current with the next. If the "$-$" point of the next intersection pair is less than the current "$-$" point, then replace the current with the next. If either the next "+" point is greater than the current "$-$" point or the next "$-$" point is less than the current "+" point, then we can conclude C_i contributes no corner points to S_i. If there is an input circle $C_j : 1 \le j < i$ that does not intersect C_i, then there are 2 cases: If C_j is contained in C_i, we again conclude C_i contributes no corner points to S_i. If C_i and C_j are disjoint, the intersection set S_n is empty. This approach requires only $O(n)$ work for each input circle, leading to an $O(n^2)$ method for finding all the corner points.

This process of using the plus/minus pairs to find the contributing corner points of C_i is illustrated in Fig. 2(ii). With two circles, there is one plus/minus pair indicating the two corner points of C_i. When a third circle is added, the "-" point is replaced. The "+" point is replaced when the fourth circle is added, and the final plus/minus pair represents the two corner points contributed by C_i.

We leave as an open question whether this can be improved to $O(n\log n)$, or even $O(n)$.

STEPS 3 and 4: Checking the Antipodal of each Corner Point

If a diameter of D_1 is contained in S_n (see Fig. 3), then D_1 is the solution to the Disk problem. Otherwise, the solution is the smallest enclosing disk containing all of the corner points. This smallest disk can be obtained by Megiddo's linear-time algorithm [13]. The checks are made in Steps 3 and 4 of the algorithm and their validity was established in [12].

Next we discuss the computational aspects of Steps 3 and 4.

Given the center of a circle (x_c, y_c) and a point on that circle (x_p, y_p), the antipodal point (x_a, y_a) can be found as $x_a = x_c + (x_c - x_p)$ and $y_a = y_c + (y_c - y_p)$.

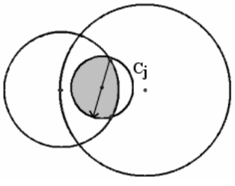


Fig. 3. Antipodal lies in S_n, thus $D_1 = D*$

To check if an antipodal point (x_a, y_a) lies in all n input disks $C_i = (x_i, y_i, r_i)$, you would simply examine if $(x_a - x_i)^2 + (y_a - y_i)^2 \le r_i^2$ is true for each $i = 1, ..., n$.

To test whether each of the $O(n)$ corner points of C_1 has an antipodal point in each of the n input disks exhaustively would require $O(n^2)$ work. Perhaps we can do better. Can we improve this step to $O(n)$? It is true that there can be $O(n)$ corner points on the smallest circle. Therefore, we would like to be able to check to see if the antipodal point of each is in the intersection, without having to explicitly check its inclusion in all the input disks.

By using the Disk Method to find $D*$, the point (x^*, y^*) is used to estimate an object's location and the localization error is bounded by $r*$.

5 The Washer Problem

We now consider the *Washer Problem.* If we can place a minimum bound, $d' - \varepsilon$, on the distance between an object and an anchor, we are sure that the object will lie outside of the disk centered at the anchor and having a radius of $d' - \varepsilon$. This forms the basis of the Washer Problem detailed below:

$O(n^3)$ Algorithm for the Washer Problem

Input: Centers and radii of n (inclusion) disks: $D_i = [x_i, y_i, r_i]$, $i = 1, \ldots, n$, where $r_1 \le r_2 \le \cdots \le r_n$, and m (exclusion) disks: $D'_i = [x'_i, y'_i, r'_i]$, $i = 1, \ldots, m$, where $r'_1 \le r'_2 \le \cdots \le r'_m$. All input disks are pair-wise distinct.

Output: Center and radius of the smallest disk $D^* = [x^*, y^*, r^*]$ that includes the set of points $S = I - E$, where $I = \left(D_1 \cap D_2 \cap \cdots \cap D_n\right)$ and $E = (D'_1 - C'_1) \cup (D'_2 - C'_2) \cup \cdots \cup (D'_m - C'_m)$. That is, S is the set of points included in all the inclusion disks and excluded from the interiors of all the exclusion disks. In Fig. 4(i), the set S is indicated by the shaded region. In the following algorithm, we assume that the number of inclusion disks and exclusion disks are approximately the same ($m \le 2n$). Therefore, $O(n) = O(m)$.

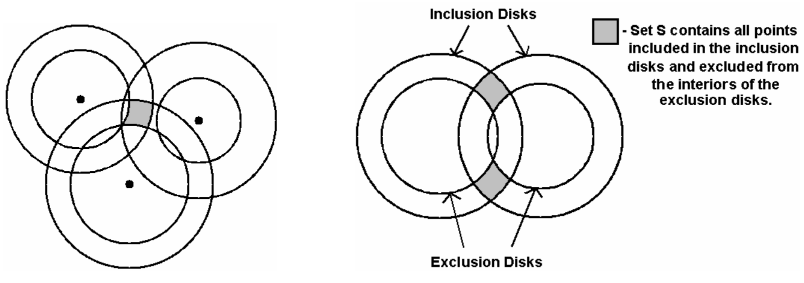

Inclusion Disks
▨ - Set S contains all points included in the inclusion disks and excluded from the interiors of the exclusion disks.
Exclusion Disks

Fig. 4. (i) the Washer problem, (ii) set S broken into multiple pieces

1. [$O(n^2)$] Find all pairwise *intersection points* of the associated circles of the n inclusion disks and m exclusion disks. There can be at most 2 intersection points for each pair of circles, for a total of $(m+n)(m+n-1)/2$ intersection points.

2. [$O(n^3)$] For each intersection point, determine if this point lies in all of the inclusion disks and outside all of the exclusion disks. Let $\{(a_k, b_k)\}$ be the set of these *corner points*.

3. [$O(n^2)$] For each of the $O(n)$ corner points that lies on the smallest input circle C_1, check to see if its antipodal point on C_1 lies in all of the inclusion disks and outside all of the exclusion disks. If any such antipodal passes this check, return C_1 as the smallest enclosing circle. Otherwise, proceed to Step 4.

4. [$O(n^2)$] Return Smallest($\{(a_k, b_k)\}$, the smallest disk containing all of the corner points $\{(a_k, b_k)\}$ using Meggido's linear-time linear programming algorithm [13]. Note that corner points ≤ $O(n^2)$.

As shown in Fig. 4(ii), it is possible that the inclusion disks and exclusion disks will break the set S into multiple disjoint pieces. Because of this, our proof of the $O(n)$ bound on the number of corner points does not carry over to the washer case. We leave open the questions of whether the number of corner points is less than $O(n^2)$ and whether they can be found in less than $O(n^3)$ time.

We can, however, still establish the validity of Step 4, which results in our $O(n^3)$ algorithm above. The proof is similar to that of the Disk Method. As with the Disk Method, the smallest enclosing circle $D*$ is used in the localization problem by estimating an object's location at (x^*, y^*) with an error bound of $r*$.

6 Performance Results

A program was developed to implement our algorithms in order to simulate the localization process under varying conditions. We used the Delphi programming environment because its excellent graphics capabilities make it ideal for illustrating visually the washers used in our technique.

The system assumes a node can estimate the distance between itself and each of its neighbors. Although more accurate ranging techniques will produce smaller localization errors, our approach is not dependent on any one ranging technique. The system also assumes that a small subset of the nodes, *anchors*, are aware of their location. Anchor nodes are either physically placed at known positions or they are equipped with a positioning technology such as GPS. Finally, for simplicity, the system assumes: (1) signals are omni directional and symmetric, (2) all nodes have the same transmission range, and (3) and the network is connected.

In our simulation experiments, we randomly deployed 500 sensor nodes over a 250,000 ft^2 region (500 x 500 foot square). The anchor nodes were placed in a mesh configuration. When running trials based on radio transmission, the radio range was assumed to be 100 feet. This range was based on experimental results using second-generation MICA2 motes [15].

When calculating perceived distance between 2 nodes, a random normally distributed error $e \in [-\varepsilon, \varepsilon]$ is generated and added to the actual distance d . The resulting perceived distance $d' = d + e$ is thus within $d \pm \varepsilon$, where ε is the maximum ranging error. Next, to ensure a correct bound, the disk radius of $d' + \varepsilon$ is used in the algorithm. When we use the Washer method, the inner disk's radius is taken to be $d' - \varepsilon$.

By having each node find an upper bound on its multi-hop distance to every anchor node, we can localize the nodes using any or all of the anchors, not just the neighboring ones. To accomplish this, each anchor node broadcasts its position to initialize the localization process. All other nodes will then broadcast these anchor positions as well as their maximum distance estimate to each anchor. After several iterations of this process, each node will now have an upper bound on the distance to every anchor node in the network. These upper bounds are used as the circles in the Disk method. If a node is a neighbor to an anchor node, it uses the distance estimate $d' - \varepsilon$ as the radius of the washer's inner circle. If the node is not a neighbor to an anchor, the best we can do is to use the maximum transmission range (e.g., 100 feet) as the washer's inner circle.

When using the exclusion disks in addition to the inclusion disks, the size of the smallest enclosing circle is sometimes reduced (see Fig. 4(i)). A node uses the distances, $d' \pm \varepsilon$, from multiple anchors to form the washers for localization. Each node chooses its closest anchors when selecting anchors to be used. In our experiments, we varied the number of anchors, or washers, used in the localization calculations from 2 to 10. A 500-node network was generated and a 6x6 mesh of 36 anchors was used. An ε of 10 feet was used for the maximum ranging error.

Fig. 5(i) shows averaged results of localizing the 500-node network. The Washer method had a slightly smaller mean actual localization error than Disk. When using 10 anchors, for example, the actual error of the Disk method was 5.29 feet and Washer was 4.60 feet. When looking at the bound on the localization error, recall that it is the radius of the smallest enclosing circle of the intersecting washers. The Washer method gave a slightly tighter bound on the error than Disk. The error bound for Disk was 14.07 feet and Washer was 10.96 feet.

First, note that the average actual error is quite small on the order of $\varepsilon / 2$, or half of the maximum ranging error. This is a bit surprising since the actual error is in 2 dimensions, whereas the ranging error is a 1-dimensional measure. Second, our error bound is also generally quite small, being approximately twice the actual error.

In Fig. 5(ii), we show how increasing the density of the anchors affects the localization error. Keeping constant the 500x500 feet region and the 100-foot radio range, we ran experiments using a 500-node network and varied the mesh sizes of anchors to be 16, 36, 64, and 100. As one might expect, the networks with more anchors per square foot produced lower errors. Having more anchors as neighbors provided smaller inclusion disks compared to the disks resulting from multi-hop paths. With 10 anchors contributing, the mean actual error for 16, 36, 64, and 100

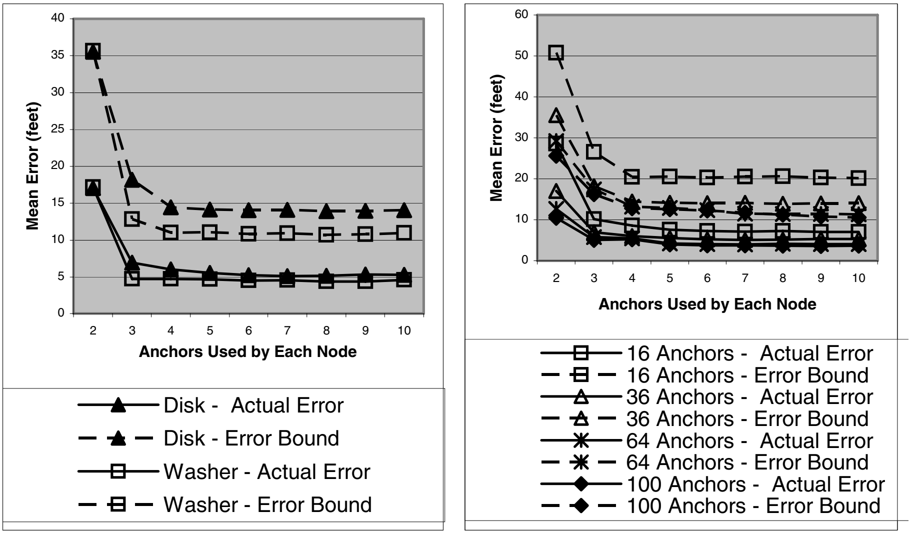


Fig. 5. (i) Disk versus Washer, (ii) Varying the number of anchors

anchors were 7.07, 5.29, 4.08, and 3.63 feet, respectively. The mean error bounds for these same networks were 20.19, 14.07, 11.33, and 10.58 feet.

In most cases, increasing the number of anchors, or washers, used for localization improved the results. This increasing of anchors used eventually plateaus. As you can see from Fig. 5(i and ii), this plateau appears to occur at about 4 or 5 anchors. As more anchors are used, however, this increases the number of computations a node must perform to localize. This can have an effect on energy utilization of a node, which in turn, affects the lifetime of a wireless sensor network. Therefore, even though we showed the efficiency for the Disk and Washer algorithms to be $O(n^2)$ and $O(n^3)$, respectively, it appears that using values of n larger than 4 or 5 will require more work but add no significant degree of accuracy.

As mentioned previously, our technique does not rely on any specific ranging technique. In Fig. 6(i), we show how the accuracy of the ranging method affects the localization error. We are using disks with radius $d'+\varepsilon$ based on distance estimates using values of 1, 10, and 30 feet for the maximum ranging error ε. In these experiments when 10 inclusion disks were used, a maximum ranging error ε of 1, 10, and 30 feet produced mean actual errors of 1.05, 5.29, and 12.66 feet respectively. The error bounds were 2.18, 14.07, and 31.72 feet. As one would expect, the smaller values of ε produce better localization accuracy. When ε of 30 feet was used, the error bound was approximately ε and the actual errors were less than $\varepsilon/2$.

In all of the previous examples, we deployed the anchors in a mesh configuration. We decided to consider an anchor configuration that places all of the anchors along the perimeter of the region. The perimeter configuration might make sense when placing anchors in a region was difficult (military, volcano, heavy foliage, etc.). We used a 500-node network of 16 anchors and compared the accuracy of a 4x4 mesh of

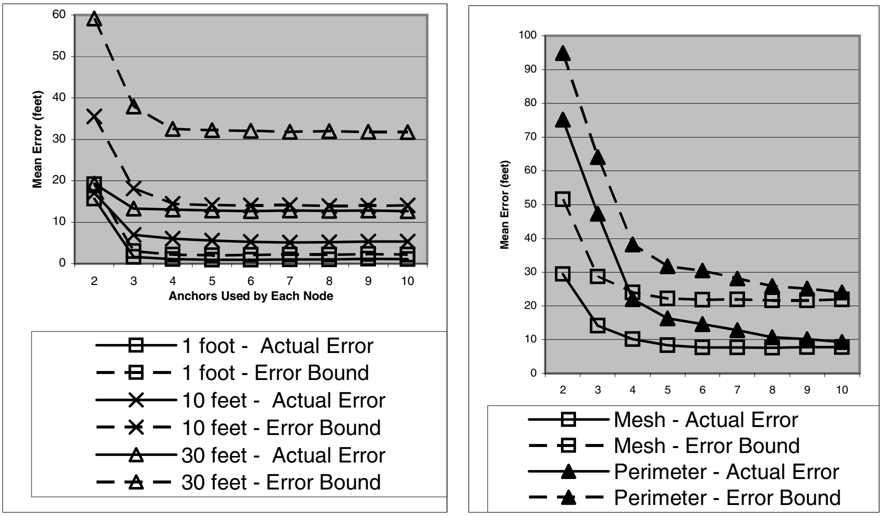


Fig. 6. (i) Varying the ranging error, (ii) Varying the number of anchors

anchors versus a perimeter arrangement with 5 anchors along each side of the region. In the perimeter scenario, over half of the nodes did not have any neighbors that were anchors.

As illustrated in Fig. 6(ii), when only a few washers were used in the computation, the perimeter configuration produced much larger errors than the mesh. As more washers contributed to the localization, however, the two configurations produced comparable results. With 10 anchors used, for example, the mean actual error was 7.76 feet for mesh and 9.31 feet for perimeter. The mean error bounds for mesh and perimeter were 21.87 and 23.97 feet, respectively. It is worth noting that when more anchors are used in the perimeter configuration, it tended to give larger errors. This was because a node would choose all of its neighboring anchors from the same boundary. The resulting smallest enclosing circle would thus be along that border instead of towards the interior of the region where the node actually lies. When using perimeter, one could address this by choosing anchors from different boundaries.

7 Conclusions and Future Work

We have shown that we can bound the error on the localization of a node by finding the smallest enclosing circle that covers the intersection of multiple disks. The disks are constructed by taking the maximum distances $d' + \varepsilon$ from anchor nodes as the disk radius and use the anchor's position as the disk center. We use the center of the smallest enclosing circle as the location estimate of the node and the radius of the circle as the error bound. We provided a novel $O(n^2)$ algorithm for finding this location estimate and error bound, where n is the number of anchors used by each node. We believe that Step 2 of our Disk algorithm in which we compute the corner points can be improved and we leave this problem as part of our future work.

We extended our Disk technique to what we called the Washer method. Based on the distance to anchors, the washers are constructed using the inclusion disks as the exterior circle and exclusion disks with radius $d' - \varepsilon$ as the inner circle. We provided an $O(n^3)$ algorithm for solving the Washer problem and are hopeful that we can improve on this efficiency in the future.

Providing a location estimate for a device is more meaningful if you can say something about the confidence that you have in the estimate. In our technique, we are saying that there must be a bound on the ranging estimate between two neighboring devices. For example, we say that there exists some maximum error ε in which distance estimates between the two devices will always fall into the range $d \pm \varepsilon$, where d is the actual distance between the devices. In some conditions, the ranging estimate may be very accurate most of the time, but occasionally produces a large error. This type of environment would yield a large value of ε and, as illustrated in Fig. 6(i), a bigger localization error. To address this, we would like to extend our work so that confidence intervals can be used in conjunction with the location estimates and error bounds.

It is important to note that our technique does not depend on the number of nodes in the network or the percentage of nodes that need to be anchors. Given a region and a set of anchor nodes in that region, our system will produce the same accuracy and amount of work per device for locating 10,000 devices as it does for locating one device. The important factors in the localization accuracy, as reported in the previous section, are anchor density, maximum ranging error, number of anchors used in the computations, and anchor configuration.

References

1. Hightower, J., Borriello, G.: Location Systems for Ubiquitous Computing. IEEE Computer, Los Alamitos (2001)
2. Szewczyk, R., Osterweil, E., Polastre, J., Hamilton, M., MainWaring, A., Estrin, D.: Habitat Monitoring with Sensor Networks. Communications of the ACM 47(6), 34–40 (2004)
3. Aspnes, J., Goldenberg, D., Yang, R.: On the computational complexity of sensor network localization. In: Nikoletseas, S.E., Rolim, J.D.P. (eds.) ALGOSENSORS 2004. LNCS, vol. 3121. Springer, Heidelberg (2004)
4. Terwilliger, M., Gupta, A., Bhuse, V., Kamal, Z., Salahuddin, M.: A Localization System using Wireless Network Sensors: A Comparison of Two Techniques. In: Proceedings of the First Workshop on Positioning, Navigation and Communication, Hannover, Germany (March 2004)
5. Terwilliger, M., Gupta, A., Khokhar, A., Greenwood, G.: Localization Using Evolution Strategies in Sensornets. In: Proceedings of the IEEE Congress on Evolutionary Computation 2005, Edinburgh, Scotland (September 2005)
6. Enge, P., Misra, P.: Special Issue on GPS: The Global Positioning System. Proceedings of the IEEE 87(1), 3–172 (1999)
7. Savvides, A., Han, C., Srivastava, M.: Dynamic Fine-Grained Localization in Ad-Hoc Networks of Sensors. In: Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking (MOBICOM 2001) (2001)

8. Savarese, C., Rabaey, J., Beutel, J.: Locationing in Distributed Ad-Hoc Wireless Sensor Networks. In: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, Salt Lake City, UT, vol. 4, p. 2037
9. Savarese, C., Rabaey, J., Langendoen, K.: Robust Positioning Algorithms for Distributed Ad-Hoc Wireless Sensor Networks. In: Proceedings of the General Track: 2002 USENIX Annual Technical Conference, pp. 317–327, June 10-15 (2002)
10. Meguerdichian, S., Slijepcevic, S., Karayan, V., Potkonjak, M.: Localized Algorithms in Wireless Ad-Hoc Networks: Location Discovery and Sensor Exposure. In: Proceedings of MobiHOC 2001, Long Beach, CA (2001)
11. Doherty, L., Pister, K., Ghaoui, L.: Convex Position Estimation in Wireless Sensor Networks. In: Proceedings of the IEEE InfoCom 2001, vol. 3, pp. 1655–1663 (2001)
12. Terwilliger, M., Gupta, A., Coullard, C.: On Bounding the Localization Errors, Technical Report TR/05-07, Department of Computer Science, Western Michigan University (2005)
13. Meggido, N.: Linear-Time Algorithms for Linear Programming in R^3 and Related Problems. SIAM Journal on Computing 12, 759–776 (1983)
14. Bourke, P.: Intersection of Two Circles (April 1997), http://astronomy.swin.edu.au/~pbourke/geom-etry/2circle/
15. Crossbow Technology (last viewed, August 2005), http://www.xbow.com

Optimization of Fast Fourier Transforms on the Blue Gene/L Supercomputer

Yogish Sabharwal[1], Saurabh K. Garg[2], Rahul Garg[1], John A. Gunnels[3], and Ramendra K. Sahoo[3]

[1] IBM India Research Laboratory, Plot-4, Block C, Vasant Kunj Institutional Area, New Delhi, India
{ysabharwal,grahul}@in.ibm.com

[2] Grid Comp. and Dist. Sys. Lab, Deptt of Comp. Sc. and Software Engg, The University of Melbourne, Australia
sgarg@csse.unimelb.edu.au

[3] IBM T. J. Watson Research Center, 1101 Kitchawan Rd, Rt. 134, Yorktown Heights, NY 10598, USA
{gunnels,rsahoo}@us.ibm.com

Abstract. We analyze the bottlenecks in the parallel FFT algorithm and describe optimizations carried out for the algorithm on the Blue Gene/L Supercomputer. We identified three avenues for improving the performance of the algorithm – single-node FFT performance, Alltoall collective performance and overlap of computation and communication. Performance at all these levels has been optimized using the double-hummer intrinsics of the Blue Gene/L CPU, careful ordering and synchronization of messages in Alltoall communications and suitable interleaving of message exchanges with computations. Using these optimizations, we obtained 20% performance improvement over the baseline version on the 64 racks Blue Gene/L system. We give a brief overview of the Alltoall optimizations, describe our computation-communication overlap strategy and present results for strong scaling and weak scaling of parallel FFT on Blue Gene/L. We also discuss the fundamental limits to scaling of the parallel transpose algorithm for computing FFT.

1 Introduction

The Discrete Fourier Transform (DFT) plays an important role in many scientific and technical applications, including time-series and waveform analysis, solutions to linear partial differential equations, convolution, digital signal processing, and image filtering [13, 15, 16]. Cooley and Tukey designed an algorithm [4] to compute the DFT of an n point series in $O(n \log n)$ operations, significantly improving over previously known methods. Many algorithms have since been proposed to compute DFTs with similar efficiency. This class of efficient algorithms is generally referred to as fast Fourier transform (FFT) algorithms. One of the most widely used FFT implementations in both academia and the

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 309–322, 2008.

industry is FFTW [7]. It provides excellent performance on a variety of machines – even competitive with or faster than equivalent libraries supplied by vendors.

Many parallel algorithms have been proposed and designed for computation of FFTs on parallel computers (see [1, 6, 17, 20] and references therein). There are two basic approaches to parallelizing FFT algorithms based on the underlying interconnection network topology – the binary-exchange algorithms and the transpose based algorithms. The former is suited for systems where the bisection bandwidth of the interconnect scales linearly with the number of nodes, whereas, the latter is more suitable for systems where the bisection bandwidth scales sub-linearly. For more details the reader is referred to [12].

Implementation of the transpose algorithm [12] carries out two basic operations - parallel computation of FFT on individual nodes with (almost) perfectly balanced load and perfectly balanced Alltoall communications where every pair of nodes exchanges the same amount of data. On a system where bisection bandwidth does not scale linearly with the number of processors, the Alltoall communication ends up taking most of the time. Moreover, as the number of processors is increased (while keeping the same problem size), the amount of data exchanged between every pair of nodes decreases quadratically. As a result, the header overheads start dominating in Alltoall communications. This leads to an absolute limit for strong as well as weak scaling of this algorithm.

There were three avenues for performance optimization of the transpose-based parallel FFT algorithm single node performance, performance of Alltoall communication and overlap of communication and communication. The single node performance was optimized using a cache efficient decomposition of FFT (using the Cooley-Tukey algorithm), Blue Gene/L specific dual floating point intrinsics and several hints to the compiler in the C code. The Alltoall performance was optimized by organizing the communication into several short phases such that each node exchanges data with exactly four other nodes in each phase. The phases were then bundled into groups which were separated using an efficient hardware-based barrier. Curiously, inserting a barrier between groups of phases improves performance by as much as 35% on the 64 racks Blue Gene/L system. This is primarily due to congestion avoidance. The computation and communication was overlapped by carefully dividing the data into smaller subsets. This results in hiding (to a large extent) the latency of the faster of the two operations behind the other. With these optimizations, we obtained a peak FFT performance of 2875 Gflops – which is the best reported FFT performance on any system built so far.

The rest of this paper is organized as follows. We discuss fast Fourier transforms and its parallel algorithms in Section 2. This is followed, in Section 3 by an analysis of the bottlenecks for the transpose based parallel FFT algorithms. The Blue Gene/L specific optimizations for single CPU performance, Alltoall communication and for overlap of computation and communication are described in Section 4. We present results for strong scaling and weak scaling of optimized parallel FFT on Blue Gene/L in Section 5. Finally, we conclude in Section 6.

2 Fast Fourier Transforms

The Discrete Fourier Transform (DFT) for a sequence of n complex numbers $x_0, ..., x_{n-1}$ is another sequence of n complex numbers $y_o, ..., y_{n-1}$, where $y_k = \sum_{j=0}^{n-1} x_j \omega^{jk}$; $k = 0, .., n-1$ and ω is the primitive nth root of unity in the complex plane, i.e., $\omega = e^{2\pi\sqrt{-1}/n}$.

Cooley and Tukey [4] presented an efficient algorithm that recursively breaks down the computations of a DFT into computation of smaller DFTs. More formally, let n be a composite and n_1, n_2 be its factors such that $n = n_1 n_2$. Rewriting the indices in the above equation as $k = k_1 \cdot n_2 + k_2$ where $0 \le k_1 < n_1, 0 \le k_2 < n_2$ and $j = j_2 \cdot n_1 + j_1$ where $0 \le j_1 < n_1, 0 < j_2 \le n_2$, we get

$$y_k = y_{(n_2 k_1 + k_2)} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x_{(n_1 j_2 + j_1)} \cdot \omega_{n_1 n_2}^{(n_1 j_2 + j_1).(n_2 k_1 + k_2)}$$

$$= \sum_{j_1=0}^{n_1-1} \left[\left(\sum_{j_2=0}^{n_2-1} x_{(n_1 j_2 + j_1)} \cdot \omega_{n_2}^{j_2 k_2} \right) \cdot \omega_n^{j_1 k_2} \right] \cdot \omega_{n_1}^{j_1 k_1} \qquad (1)$$

Therefore, the DFT of size n can now be evaluated by first computing n_1 DFTs of size n_2, multiplying by twiddle factors (complex roots of unity) and then computing n_2 DFTs of size n_1. Many algorithms [2, 3, 9, 18, 19] have since been proposed, that achieve similar efficiency. This class of algorithms are referred to as fast Fourier transform (FFT) algorithms.

The transpose based parallel FFT algorithms are designed for systems with interconnects that do not scale linearly with the number of nodes. These algorithms consider the input as a logical multi-dimensional matrix and break down the DFT computations using Cooley-Tukey algorithm into smaller DFT computations operating along each of the dimensions of the matrix. These algorithms then iterate over the different dimensions of the matrix computing DFTs along those dimensions. In order to ensure that the data required for the DFT computations in the current iteration lie on the same processor, matrix-transpose operations are required along two-dimensional planes of the matrix. Hence, the name transpose FFT algorithms. The simplest form of the transpose FFT algorithm is the two-dimensional FFT algorithm. This is described in Section 2.1. The FFTE algorithm is a variant of the transpose algorithm (c.f. Section 2.2).

2.1 Two-Dimensional Transpose Algorithm

The 2D-transpose FFT algorithm considers the input vector as a logical two-dimensional matrix and breaks down the DFT into sub-steps based on the Cooley-Tukey algorithm. The main idea behind the algorithm is to logically express the vector, x, of size N, whose DFT is required to be computed, as a two-dimensional matrix of dimension $n_1 \times n_2$, where $n = n_1 \cdot n_2$. The Cooley-Tukey algorithm (see equation 1) can now be used to recursively compute the DFT of x.

Algorithm 1. Two-dimensional Transpose FFT Algorithm

```
1. Distribute columns on each processor - Global Transpose (Alltoall)
2. Compute $\sqrt{n}/p$ DFTs of size $\sqrt{n}$ along the matrix columns
3. Distribute rows on each processor - Global Transpose (Alltoall)
4. Multiply by twiddle factors and Rearrange
5. Compute $\sqrt{n}/p$ DFTs of size $\sqrt{n}$ along the matrix rows
6. Rearrange the output data elements - Global Transpose (Alltoall)
```

A high level description of the 2D-Transpose FFT algorithm is presented in Algorithm 1. For simplicity of exposition, we assume that n is an even power of 2 and $n_1 = n_2 = \sqrt{n}$. The data can therefore be considered to form a $\sqrt{n} \times \sqrt{n}$ matrix. The transpose algorithm stripes the matrix into $\sqrt{n}/p$ columns on each processor. Each processor computes $\sqrt{n}/p$ DFTs of size $\sqrt{n}$ (along each column of the matrix). All processors then performs the twiddle factor multiplications on their local data. The next phase is to perform the DFTs along each row of the matrix. However, since the rows of the matrix are distributed amongst the processors, the algorithm first performs a matrix-transpose in order to transform the rows into columns that reside on the same processor. Each processor then performs the $\sqrt{n}/p$ DFTs of size $\sqrt{n}$ (along the new columns).

Typically the input vector is initially distributed over the processors in a way such that n/p contiguous elements of the vector lie on each processor – this can also logically be viewed as a two-dimensional $\sqrt{n} \times \sqrt{n}$ row-major matrix with $\sqrt{n}/p$ rows distributed on each processor. Therefore an additional matrix transpose is required in order to have the columns distributed amongst the processors as required by the transpose algorithm. Similarly, another transpose is required at the end in order to rearrange the output vector so that the contiguous elements reside on the same processor. The matrix transpose can be performed by using the Alltoall collective in conjunction with local transpose of blocks received from each processor (or sent to each processor).

2.2 FFTE Parallel Algorithm

The FFTE algorithm is a variant of the Transpose algorithm which is more suited for vector processors. In this algorithm, the input vector is logically considered as a two-dimensional matrix. For simplicity of exposition, we assume the matrix dimensions to be $n^{2/3} \times n^{1/3}$. There are three Alltoall phases as before. In the first computation phase, each processor computes $n^{2/3}/p$ DFTs, each of size $n^{1/3}$. In the second computation phase, there is a small variation. Each processor, instead of computing $n^{1/3}/p$ DFTs of size $n^{2/3}$ each, breaks down each of these computations in a manner similar to the two-dimensional transpose algorithm itself. Therefore It considers each vector of size $n^{2/3}$ as a matrix of size $n^{1/3} \times n^{1/3}$ and applies the Transpose algorithm again. Note however, that each of these $n^{1/3} \times n^{1/3}$ matrices lie on a single processor; hence the matrix transpose is a local transpose operation – not requiring any communication between the nodes.

3 Bottleneck Analysis

As described in the previous Section, parallel FFT algorithms perform FFT computations in five phases, with three phases of global transpose (Alltoall communication) interleaved with two phases of parallel single-node FFTs. If there is no computation-communication overlap the time taken for performing the FFT can be broken down into two parts – the computation time (t_{comp}) and the communication time (t_{comm}): $t_{fft} = t_{comp} + t_{comm}$.

3.1 Computation Time

In each computation phase, a node performs one dimensional FFT computations on a set of vectors present in the local memory. The number of floating point operations needed to compute FFT of a vector of size n is $5n\log(n)$. The time taken for this computation is $5c \cdot n\log n$, where $1/c$ is the floating point performance (FLOPS) of the single-node FFT implementation on the architecture.

Consider the 2D-transpose FFT algorithm. For a vector of size n distributed over p processors, in each of the two computation phases, each node computes $\sqrt{n}/p$ DFTs each of size $\sqrt{n}$. Therefore the parallel computation time is

$$t_{comp} \;=\; 5 \cdot 2\frac{\sqrt{n}}{p} \cdot c\sqrt{n}\log\sqrt{n} \;=\; 5\; c \cdot \frac{n}{p} \cdot \log n$$

The computation time is inversely proportional to the number of nodes.

3.2 Communication Time

In this subsection we discuss the communication performance of the parallel 2-dimensional transpose-FFT algorithm, implemented on a system with a d-dimensional torus network. We first discuss strong and weak scaling limits when the communication and header overheads are ignored and then we modify the analysis take the overheads into account.

Performance without header overheads. Consider the FFT computation of a vector of size n distributed over p processors. Let b be the size (in bytes) of each element of the vector and B be the bisection bandwidth of the underlying interconnection network in bytes-per-second. All the data (ignoring self-transfers) is exchanged in each phase of the Alltoall communication. Note that one-fourth of the data would cross any bisection in one direction. Therefore, the communication time for performing the Alltoall collective is at least $t_{a2a} = nb/(4B)$.

For a symmetric d-dimensional torus network with link bandwidth l and $p^{1/d}$ processors in each dimension, the bisection bandwidth is $B = 2P^{(d-1)/d}l$. Therefore, the communication time for performing three Alltoall collective on such a system is $t_{comm} = 3nb/(8 \cdot l \cdot p^{(d-1)/d})$. For the more generic case, when the dimensions of the torus are not equal it is $t_{comm} = 3nb/(8 \cdot l \cdot (p/p_m))$ where p_m is the size of the longest dimension. Thus, the performance of parallel FFT based on a two-dimensional transpose algorithm is

$$FLOPS = \frac{5p}{5c + \frac{3}{8}\frac{b}{l}\frac{p^{1/d}}{\log(n)}} \qquad (2)$$

Strong Scaling. For determining strong scaling, the problem size (n) is kept fixed and the number of processors (p) is varied. For small values of p (when $\frac{l/b}{1/c} >> \frac{3}{40}\frac{p^{1/d}}{\log(n)}$) the scaling is linear as $FLOPS \approx p/c$. Note that the ratio $(l/b)/(1/c)$, called "flops-to-bandwidth ratio", represents the ratio of time taken to transfer one floating-point element to the time taken to perform one floating-point operation. As p is increased, the second term in denominator of equation 2 begins to dominate the first term (when $p >> \left[40/3\left(\frac{l/b}{1/c}\right)\log(n)\right]^{d}$). In this case the FFT performance is

$$FLOPS \approx \frac{40}{3}\frac{l}{b}\log(n)p^{(d-1)/d}$$

In the transpose algorithm, the Alltoall size (data exchanged between a pair of nodes) is nb/p^2. Therefore, the maximum value of p is equal to $\sqrt{n}$ which gives a maximum performance of

$$FLOPS \approx \frac{40}{3}\frac{l}{b}n^{(d-1)/2d}\log(n)$$

Weak Scaling. In this case, the problem size per node is kept constant as the number of processors is varied. Thus $n = pM/b$ where M is per-node memory allocated to the problem. Considering $p^2 \leq n$, the maximum value p can take is M/b. Therefore, the maximum achievable performance under weak scaling is

$$FLOPS = \frac{5M}{5bc + \frac{3}{16}\frac{b^{(2d-1)/d}}{l}\frac{M^{1/d}}{\log(M/b)}}$$

Including header overheads. Considering header overheads, the amount of data exchanged between every pair of nodes is

$$\frac{nb}{p^2} + \left\lceil \frac{nb}{p^2 D_{max}} \right\rceil h$$

where, h is the overhead per packet in bytes (including header, trailer, acknowledgments, etc.), and D_{max} is the largest payload size (in bytes) that can be sent in a packet. In case $nb/p^2 >> D_{max}$, the amount of data exchanged between any pair of nodes can be approximated as $\frac{nb}{p^2}(1+\frac{h}{D_{max}})$. In this case the performance of the FFT algorithm will be same as before with b replaced by $b(1+\frac{h}{D_{max}})$.

In the case when $\frac{nb}{p^2} \leq D_{max}$, the amount of data exchanged between all the node pairs is $\frac{nb}{p^2} + h$. Therefore, the total communication time is

$$t_{comm} = \frac{3}{4B}\cdot p^2\left[\frac{nb}{p^2} + h\right] = \frac{3}{8}\frac{(nb + p^2 h)}{p^{(d-1)/d}l}.$$

The FFT performance in this case is

$$FLOPS = \frac{5p}{5c + \frac{3}{8}(\frac{b}{l} + \frac{hp^2}{nl})\frac{p^{1/d}}{\log(n)}} \tag{3}$$

Strong Scaling. Recall that for the transpose algorithm, $p^2 \leq n$. So, if the header overhead, h, is significantly larger than the size of a vector element, b, and $\frac{l/h}{1/c} =<< \frac{3}{40} \frac{p^{1/d}}{\log{(n)}}$, then the numerator scales as $O(p)$ whereas the denominator scales as $O(p^{2+1/d})$. Therefore, in this range, increasing the number of processors will degrade performance.

Weak Scaling. In this case, $n = Mp/b$ and also $p^2 \leq n$. Therefore, under the conditions stated above, the numerator scales as $O(p)$ while the denominator scales as $O(p^{1+1/d})$. Even in this case, there is an absolute limit to which performance may scale.

The absolute performance limits can be estimated from equation 3.

4 Optimizations to Parallel FFT

There are three avenues for optimization of the FFT algorithm – (i) decrease the computation time, (ii) decrease the communication time, i.e. improve Alltoall performance and (iii) overlap the computation with communication. We describe how we optimized the parallel FFT algorithm on the Blue Gene/L Supercomputer, using all the three optimizations. Since Blue Gene/L is based on a 3-dimensional torus network, the bottleneck for FFT switches from computation to communication as the number of nodes is increased. Therefore, all the optimizations are required for good performance over the full range of Blue Gene/L system sizes. We start with a brief overview of the Blue Gene/L Supercomputer followed by a discussion on our optimizations in the following subsections. The best FFT performance know till date is using the HPC Challenge benchmark [5]. We use this benchmark to study the the effect of our optimizations.

4.1 Blue Gene/L Overview

The Blue Gene/L is a massively parallel supercomputer that scales up to 104K dual-processor nodes [8]. Each node has two embedded 700 MHz `PPC440` processor cores. The Blue Gene/L uses five interconnect networks, the most significant of which is the three-dimensional torus that has the highest aggregate bandwidth and handles the bulk of all communication. Each node supports six independent 1.4 Gbps bidirectional nearest neighbor links, with an aggregate bandwidth of 2.1 GB/s. The torus network uses both dynamic (adaptive) and deterministic routing with virtual buffering and cut-through capability. The messaging is based on variable size packets, each $n \times 32$ bytes, where $n = 1$ to 8 "chunks". The first eight bytes of each packet contain link-level information, routing information and a byte-wide cyclic redundancy check (CRC) that detects header data corruption during transmission. In addition, a 32-bit trailer is appended to each packet that includes a 24-bit CRC.

4.2 Single-Node Performance

In order to utilize the SIMD load-store and FMA units, we replaced the C code with Blue Gene/L intrinsics, which have a one-to-one correspondence with assembly instructions. These instructions are not identical to assembly instructions because the individual instructions are still scheduled by, and the registers allocated/managed by, the compiler. This is advantageous in terms of productivity and, typically, the compiler does a good job with register allocation and scheduling. However, there are disadvantages to this arrangement that one can address in a number of different ways.

Because the compiler handles the register allocation, there can be register spills. While the compiler spills and restores in a very efficient manner, these spills do take up execution slots that could be used for other purposes and they can degrade performance. This can be addressed in two ways. First, it is our experience that reducing the number of pseudo registers asked of the compiler, the spill rate will go down. Second, we could replace the intrinsics with assembly instructions and handle the register allocation ourselves.

Further, the compiler appears to schedule instructions as if the data of interest was resident in the L1 data cache. During FFT operations this is not always the case. Unfortunately, the appropriateness of this approximation made to reality (by the compiler) becomes apparent when the remedies are considered. While it is possible to schedule the instructions in assembler (or by turning off the scheduler and using intrinsics) so as to allow a longer load to use latency (12 cycles for covering L2 latency instead of 5 for covering L1 latency), there are resource limitations that force this strategy to yield limited returns. The cores of the Blue Gene/L system are limited to handling 4 outstanding (non-L1) loads in 3 different cache lines. By arranging the strides of the load so consecutive loads address different cache lines (and loading the following quad-words in the shadow of the L1 load), it is possible to improve the covered latency by approximately 50%, but further improvements appear difficult. It should be pointed out that the L2 prefetch unit makes this limitation considerably less problematic than it would be on a system not so equipped.

4.3 Alltoall Collective Algorithm

As shown in Section 3, the performance of the Alltoall communication collective is bandwidth bound on systems such as Blue Gene/L, where the bisection bandwidth does not scale linearly with the number of nodes. Therefore congestion build-up can have adverse effects on the performance of the Alltoall performance. In this Section, we discuss an algorithm for the Alltoall collective, called *barrier-synchronization* algorithm that we proposed [11] for the Alltoall collective, specially designed for the torus-interconnect. This Alltoall algorithm gives significant improvements over the Blue Gene/L product MPI Alltoall – up to 35% on 64K systems. We briefly summarize the approach here for completeness.

The main idea behind the barrier-synchronization algorithm is to periodically clear up any congestion that may have built up in the communication network by draining the network completely. This eliminates long term effects of congestion build-ups. In order to do this, we divide the Alltoall communication into multiple phases and require these phases to exhibit certain properties:

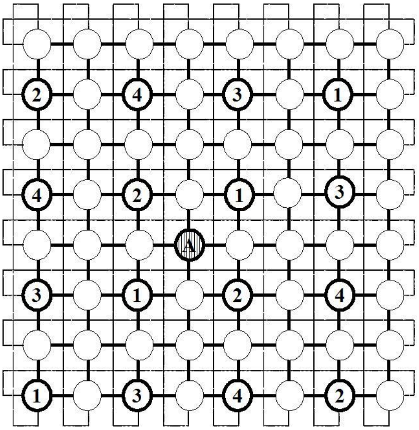


Fig. 1. Communication patterns (for sender A) in different phases on a 2D 8x8 torus

Load-balancing of links: We ensure that in each phase, the load on all the links of the 3D-torus interconnect is the same. This is to avoid local hot-spots as far as possible.

Load-balancing of phases: We ensure that in each phase, the total traffic load (in terms of byte-hops) is the same.

Synchronization: We group these phases and then separate them with the fast Blue Gene/L hardware barrier. This has two advantages. First, it ensures that the network is completely drained after each phase – so that congestion effects are not carried over across the grouped phases. Second, it prevents network buffers from becoming full by limiting the total number of packets entering the network. The number of phases to be grouped together is determined by the amount of data to be exchanged with each node and the network buffers available on the Blue Gene/L.

In order to ensure good load-balancing of links, it is very important to decide which other nodes a given node will communicate with in each iteration. Consider a 3D-torus of dimension $p_x \times p_y \times p_z$. In each phase, a node (x, y, z) sends data to exactly four other nodes given by[1] $(x+i, y+j, z+k)$, $(x-i, y-j, z-k)$, $(x+p_x/2-i, y+p_y/2-j, z+p_z/2-k)$ and $(x-p_x/2+i, y-p_y/2+j, z-p_z/2+k)$. The values of i, j, k are chosen in phases such that the full space of nodes is spanned. It can be verified that except in the cases $(i, j, k) \in \{(0, 0, 0), (\pm P_x/2, \pm P_y/2, \pm P_z/2)\}$: (a) the four nodes are always distinct (b) load across bottleneck links is perfectly balanced and (c) all phases have the same load (byte-hops). The phases corresponding to $(i, j, k) \in \{(0, 0, 0), (\pm P_x/2, \pm P_y/2, \pm P_z/2)\}$ require special handling but are few and do not take significant time. Figure 1 shows the traffic pattern for node A in a 2 dimensional 8x8 torus network. One may verify that with this scheme, the links in each dimension as well as the phases are perfectly balanced.

[1] Addition and subtraction along dimensions x, y, z are modulo p_x, p_y, p_z respectively.

Algorithm 2. Block-vector Transpose FFT Algorithm Pseudo-code

```
1. Distribute columns on each processor - Global Transpose (Alltoall)
2. /* Compute DFTs of matrix columns and overlap with Alltoall */
   a. Compute DFTs of 1st block of column vectors, i.e., $1^{st}$ $\sqrt{n}/pm$ columns
   b. For $j = 1$ to $m - 1$ do
      i.  Initiate DFTs of $(j + 1)^{th}$ block of column vectors on $2^{nd}$ core
      ii. Distribute $j^{th}$ block of column vectors using Alltoall
   c. Distribute $m^{th}$ block of column vectors using Alltoall
3. Multiply by twiddle factors and Rearrange
4. /* Compute column DFTs and overlap with Alltoall */ Similar to Step 2
5. Rearrange the output data elements - Global Transpose (Alltoall)
```

We implemented our Alltoall algorithm directly using the underlying lower layer communication primitives.

4.4 Computation Communication Overlap

The partitioning of parallel FFT algorithms into computation and communication phases makes it an attractive option to overlap computation and communication. This requires careful partitioning of data into smaller data sets which are independent of each other.

For this optimization, we replace the FFTE algorithm with the simpler two-dimensional Transpose FFT algorithm. It is easier to conceptualize and implement the computation-communication overlap for this simpler case. Although this is not the most efficient implementation for FFT, it allows us to achieve the overlap more easily.

Note that the $\sqrt{n}/p$ DFT computations involved in step 2 of the transpose FFT algorithm (Algorithm 1) along each column are independent of each other. Similarly the DFT computations in step 5 along each row are also independent of each other. Therefore, it is possible to overlap the computations of phases 2 and 5 with the communication of phases 3 and 6 respectively. The optimized algorithm incorporating computation-communication overlap is illustrated in Algorithm 2. In order to achieve the overlap, the $\sqrt{n}/p$ vectors on each processor in each phase are divided into m blocks of $b = \sqrt{n}/pm$ vectors each. The Alltoall communication of phase 3 and 6 are also split into m Alltoall communication calls each. The computation of the $(j + 1)^{th}$ block of vectors can now be overlapped with the Alltoall communication of the j^{th} block of vectors (already computed in the previous iteration).

When the computation and communication are divided into m sub-phases, the smaller of the two costs (computation and communication) gets hidden behind the other, except for one iteration of pre-processing/post-processing that is not overlapped. Therefore, the total time taken by the FFT algorithm reduces to $max\{(1/m)t_{comp} + t_{comm}, t_{comp} + (1/m)t_{comm}\}$, where the first term corresponds to the case where communication is the bottleneck and the second term corresponds to the case where computation is the bottleneck.

Some of these overlap optimizations have been previously discussed in [10] – however there, the authors proposed the use of asynchronous send and receive operations for performing the communication. This rules out the use of optimized Alltoall collectives for which many optimization techniques can be applied (as described in the previous section) therefore obtaining much better performance for the communication phase in comparison to asynchronous send/receives. We propose techniques for overlapping computation and communication that still use the Alltoall collective which can be implemented on systems supporting multiple threads – with one thread being dedicated to computation and the other to Alltoall communication. This technique lends itself naturally to the Blue Gene/L system that has dual-core nodes whereby one of the cores can be dedicated to computation and the other to Alltoall communication.

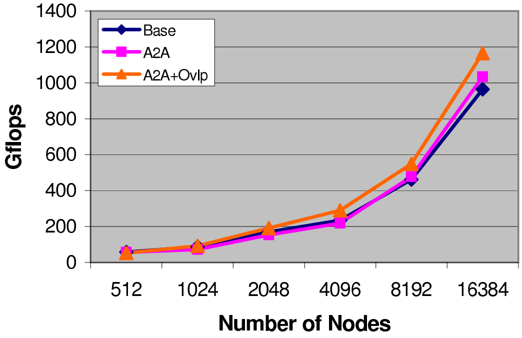
On Blue Gene/L, the dual-core nodes are used in the overlapping of computation and communication. The main thread executes on core 0 (master-core). It allocates the work of DFT computations to core 1 in Step 2.b.i. (and Step 4.b.i.) of the algorithm above. After initiating the DFT computations, it proceeds to perform the Alltoall communication for the already computed DFTs (Steps 2.b.ii. and 4.b.ii.). Once it completes the communication, it waits on core 1 to complete the computations, so that it can proceed to the next iteration. As mentioned in Section 4.1, the caches of the dual-core nodes on Blue Gene/L are not coherent. Therefore special care needs to be taken to flush/invalidate the caches. We flush the cache on core 0 before initiating work on core 1 and we flush the cache on core 1 after it completes the computations. This ensures that there is no stale data in either of the caches.

5 Performance Results

In this Section, we discuss the performance of parallel FFT obtained with our optimizations. We present weak scaling and strong scaling results.

Weak Scaling. Figure 2 compares the performance of the FFT algorithm with various optimizations. The *base* algorithm refers to the HPC Challenge FFT algorithm in which the single node FFT code is optimized to use the double hummer features of Blue Gene/L. The *A2A* algorithm refers to the base algorithm with the Alltoall collective modified to use the optimizations mentioned in Section 4.3. Finally, the *A2A+Ovlp* algorithm refers to the base algorithm modified to include both – the Alltoall optimizations and the computation-communication overlap optimizations. The size of the vectors in these runs are determined such that the vector size per-node ranges between 2M to 4M elements per node and the total vector size is a perfect square.

It can be seen that the performance of A2A algorithm is a little worse than the base algorithm but improves with increasing number of nodes. This is expected because the proposed Alltoall algorithm introduces barrier overheads to clear up the network. These overheads degrade performance for small sizes. The Alltoall optimizations are intended to regularly clear up congestion hot-spots, which impact performance much more for large number of nodes. The performance of the



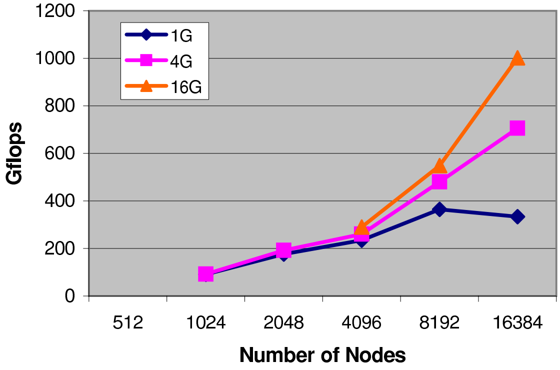
Num Nodes	Base (Gflops)	A2A (Gflops)	A2A+Ovlp (Gflops)
512	58	57	53
1024	87	72	92
2048	169	154	192
4096	235	219	290
8192	462	478	549
16384	964	1034	1166
65536	2311	2875	2779

Fig. 2. Comparison of base FFT algorithm with algorithm incorporating optimized Alltoall (A2A), and both Alltoall and computation-communication overlap (A2A+Ovlp)

optimized algorithm with both optimizations improves over the base algorithm by as much as 20% for large systems.

On the 64K node Blue Gene/L system, the FFT time is dominated by the Alltoall communication time. For a vector of size 274877906944 of double precision complex numbers ($b = 16$), the base algorithm performs the FFT in 22.6 seconds at a rate of 2311 Gflops. The time spent in the computation and communication phases is 5.5 seconds and 17.1 seconds respectively. More than 75% of the time is spent in performing global Alltoall collectives. Our calculations indicate that this Alltoall performance is 63% of the theoretically achievable peak performance (after factoring in header and other overheads). Our Alltoall algorithm takes 4.1 seconds for the same communication – about 85% of the theoretically achievable peak performance. This gives a performance of 2875 Gflops for the FFT algorithm using the Alltoall optimizations alone, which is a 20% improvement over the base algorithm. The performance does not improve using computation-communication overlap optimizations along with the Alltoall optimizations (2779 Gflops). The reason is that the overlap techniques result in dividing the Alltoall into multiple phases as well - resulting in an Alltoall size of 256 bytes in each subphase. Therefore $nb/p^2 < D_{max}$, resulting in overheads related to header, trailer and acknowledgments becoming relatively costly (See Section 3.2). Hence, the overlapping techniques are useful only if the Alltoall size is sufficiently large.

Strong Scaling. In Figure 3, we compare the FFT performance for different vector sizes ranging from 1G to 16G elements over different number of nodes. Performance scales well with increasing number of nodes. However, after scaling to a certain number of nodes, the performance starts to drop. This can be observed for the case of 1 GigaElements vector size, where the performance drops on 64K nodes. This is because the Alltoall size drops to $nb/p^2 = 2^{30} \cdot 16/16348^2 = 64$ bytes for this case. This is much smaller than the maximum data transferable per packet which is $D_{max} = 240$ bytes. Therefore, as discussed in Section 3.2, the header/trailer and acknowledgment overheads become large compared to the



Num	Number of elements		
Nodes	1 Gig	4 Gig	16 Gig
1024	91	92	-
2048	176	192	-
4096	235	260	290
8192	364	480	549
16384	334	706	1002

Fig. 3. Performance of optimized FFT algorithm for different vector sizes

data being transferred, resulting in inefficient use of the communication network. Hence, performance scales well as long as the Alltoall size $nb/p^2 > D_{max}$.

6 Conclusions

In this paper, we analyzed the performance of two-dimensional transpose-based FFT algorithm on a massively parallel system with a d-dimensional torus network. On the Blue Gene/L Supercomputer, which is based on a 3-dimensional torus network, we identified the performance bottlenecks and optimized the performance by (i) improving the performance of the single-node FFT algorithm; (ii) improving the Alltoall algorithm and (iii) overlapping computation and communication. These optimization resulted in significant performance improvement.

Techniques discussed in this paper are also applicable to parallel transpose-based algorithms for computing 2D and 3D-FFT. These algorithms also use single-node FFT computations and Alltoall communication. In these cases, the Alltoall is generally restricted to smaller communicators (a subset of the nodes). On torus interconnects, these communicators are typically mapped to planes of the torus. Therefore the Alltoall optimizations can still be applied in two-dimensions. Finally, the computation-communication overlap techniques are also applicable as these algorithms also perform computations and matrix transpositions in separate phases.

Acknowledgments

We thank James Sexton for closely working with us on the HPC Challenge benchmarks. We thank Sameer Kumar and Philip Heidelberger for useful discussions related to the low-level communication library for Blue Gene and Alltoall optimizations.

References

1. Agarwal, R.C., Gustavson, F.G., Zubair, M.: A high performance parallel algorithm for 1-D FFT. In: Proceedings of the IEEE Conference on Supercomputing, pp. 34–40 (1994)
2. Brenner, N., Rader, C.: A New Principle for Fast Fourier Transformation. IEEE Acoustics, Speech & Signal Processing 24, 264–266 (1976)
3. Bruun, G.: z-Transform DFT filters and FFTs. IEEE Trans. on Acoustics, Speech and Signal Processing, 56–63 (1978)
4. Cooley, J.W., John, W.: Tukey: An algorithm for the machine calculation of complex Fourier series. Math. Comput. 19, 297–301 (1965)
5. Dongarra, J., Luszczek, P.: Introduction to the HPC Challenge benchmark suite. TR ICL-UT-05-01, ICL (2005)
6. Eleftheriou, M., Moreira, J.E., Fitch, B.G., Germain, R.S.: A Volumetric FFT for BlueGene/L. In: International Conference on High Performance Computing, pp. 194–203 (2003)
7. Frigo, M., Johnson, S.G.: The Design and Implementation of FFTW3. Invited paper, Special Issue on Program Generation, Optimization, and Platform Adaptation. Proceedings of the IEEE 93(2), 216–231 (2005)
8. Gara, A., Blumrich, M.A., Chen, D., Chiu, G.L.-T., Coteus, P., Giampapa, M.E., Haring, R.A., Heidelberger, P., Hoenicke, D., Kopcsay, G.V., Liebsch, T.A., Ohmacht, M., Steinmacher-Burow, B.D., Takken, T., Vranas, P.: Overview of the Blue Gene/L system architecture. IBM Journal of Research and Development 49, 195–212 (2005)
9. Good, I.J.: The interaction algorithm and practical Fourier analysis. J. R. Statist. Soc. B 20(2), 361–372 (1958); Addendum, ibid. 22(2), 373-375 (1960)
10. Gupta, S.K.S., Huang, C.-H., Sadayappan, P., Johnson, R.W.: A technique for overlapping computation and communication for block recursive algorithms. Concurrency: Practice and Experience 10(2), 73–90 (1998)
11. Kumar, S., Sabharwal, Y., Garg, R., Heidelberger, P.: Performance Analysis and Optimization of All-to-all communication on the Blue Gene/L Supercomputer. In: Proc. of the IEEE International Conference on Parallel Processing (2008)
12. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing: Design and Analysis of Algorithms. Benjamin-Cummings Publishing Co., Inc. (1994)
13. Perry, M.: Using 2-D FFTs for object recognition. In: ICSPAT, DSP World Expo., pp. 1043–1048 (1994)
14. Rader, C.M.: Discrete Fourier transforms when the number of data samples is prime. Proc. IEEE 56, 1107–1108 (1968)
15. Sridharan, S., Dawson, E., Goldgurg, B.: Speech encryption using discrete orthogonal transforms. In: ICASSP, pp. 1647–1650 (1990)
16. Stearns, S.D., David, R.A.: Signal Processing Algorithms. Prentice Hall, Englewood Cliffs (1993)
17. Takahashi, D.: High-performance parallel FFT algorithms for the HITACHI SR8000. In: Proceedings of the Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, vol. 1, pp. 192–199 (2000)
18. Thomas, L.H.: Using a computer to solve problems in physics. In: Applications of Digital Computers (1963)
19. Winograd, S.: On computing the discrete Fourier transform. Math. Computation 32, 175–199 (1978)
20. MPI FFTW, http://www.fftw.org/fftw2_doc/fftw_4.html

ScELA: Scalable and Extensible Launching Architecture for Clusters[*]

Jaidev K. Sridhar, Matthew J. Koop, Jonathan L. Perkins,
and Dhabaleswar K. Panda

Network-Based Computing Laboratory
The Ohio State University
2015 Neil Ave., Columbus, OH 43210 USA
{sridharj,koop,perkinjo,panda}@cse.ohio-state.edu

Abstract. As cluster sizes head into tens of thousands, current job launch mechanisms do not scale as they are limited by resource constraints as well as performance bottlenecks. The job launch process includes two phases – spawning of processes on processors and information exchange between processes for job initialization. Implementations of various programming models follow distinct protocols for the information exchange phase. We present the design of a scalable, extensible and high-performance job launch architecture for very large scale parallel computing. We present implementations of this architecture which achieve a speedup of more than 700% in launching a simple *Hello World* MPI application on $10,240$ processor cores and also scale to more than 3 times the number of processor cores compared to prior solutions.

1 Introduction

Clusters continue to increase rapidly in size, fueled by the ever-increasing computing demands of applications. As an example of this trend we examine the Top500 list [1], a biannual list of the top 500 supercomputers in the World. In 2000 the largest cluster, ASCI White, had $8,192$ cores. By comparison, last year the top-ranked BlueGene/L had over $200,000$ cores. Even as clusters increase in node counts, an emerging trend is increase in number of processing cores per node. For instance, the Sandia Thunderbird [2] cluster introduced in 2006 has $4K$ nodes – each with dual CPUs for a total of $8K$ processors, while the TACC Ranger cluster introduced in 2008 has $4K$ nodes – each with four quad-core CPUs for a total of $64K$ processors.

Programming models and their scalability have been a large focus as cluster size continues to increase. In addition to these concerns, other more basic concerns with regard to the system software must also be addressed. In particular,

[*] This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CNS-0403342 and #CCF-0702675; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Intel, Mellanox, Cisco, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Advanced Clustering, Appro, QLogic, and Sun Microsystems.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 323–335, 2008.

the mechanism by which jobs are launched on these large-scale clusters must also be examined. All programming models require some executable to be started on each node in the cluster. Others, such as the Message Passing Interface (MPI) [3], may have multiple processes per node – one per core. Our work shows that current designs for launching of MPI jobs can take more than 3 minutes for $10,000$ processes and have trouble scaling beyond that level.

In this paper we present a scalable and extensible launching architecture (ScELA) for clusters to address this need. We note that the initialization phase of most parallel programming models involve some form of communication to discover other processes in a parallel job and exchange initialization information. Our multi-core aware architecture provides two main components: a scalable spawning agent and a set of communication primitives. The spawning agent starts executables on target processors and the communication primitives are used within the executables to communicate necessary initialization information. As redundant information is exchanged on multi-core systems, we design a hierarchical cache to reduce the amount of communication.

To demonstrate the scalability and extensibility of the framework we redesign the launch mechanisms for both MVAPICH [4], a popular MPI library, and the Process Management Interface (PMI), a generic interface used by MPI libraries such as MPICH2 [5] and MVAPICH2 [6]. We show that ScELA is able to improve launch times at large cluster sizes by over 700%. Further, we demonstrate that our proposed framework is also able to scale to at least $32,000$ cores, more than three times the scalability of the previous design.

Although our case studies use MPI, ScELA is agnostic as to the programming model or program being launched. We expect other models such as Unified Parallel C (UPC) [7] to be able to use this architecture as well. In addition, ScELA can be used to run commands remotely on other nodes in parallel, such as simple commands like 'hostname' or maintenance tasks. It is a generic launching framework for large-scale systems.

The remaining parts of this paper are organized as follows: In Section 2 we describe the goals and design issues of our launch framework. We use our framework to redesign two job launch protocols and present these case studies in Section 3. Section 4 contains a performance evaluation of the ScELA design. Related work is discussed in Section 5. We conclude and give future directions in Section 6.

2 Proposed Design

In this section we describe the ScELA framework. The main goals of the design are scalability towards a large number of processing cores, ease of extensibility and elimination of bottlenecks such as network congestion and resource limits. For ease of extensibility the various components of ScELA are divided into distinct layers. Figure 1 shows an overview of the framework. The following sections describe each of these layers in detail.

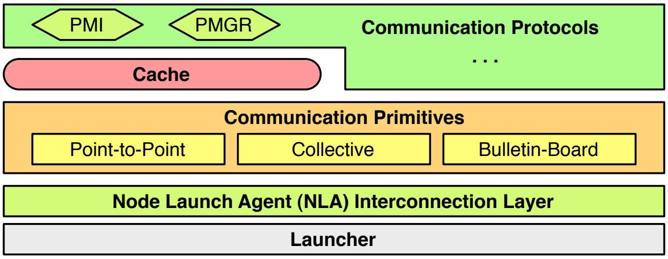


Fig. 1. ScELA Framework

2.1 Launcher

The launcher is the central manager of the framework. The job-launch process starts with the launcher and it is the only layer that has user interaction. The main task of the launcher is to identify target nodes, set up the runtime environment and launch processes on the target nodes.

Process Launching. Modern clusters deploy multi-core compute nodes that enable multiple processes to be launched on a node. On such systems, a launcher would have to duplicate effort to launch multiple processes on a node. ScELA has a Node Launch Agent (NLA) which is used to launch all processes on a particular node. The launcher establishes a connection to target nodes and sets up a NLA on each of them. This mechanism allows the Launcher to make progress on launching processes on other nodes while local NLAs handle node level process launching. The NLAs are active for the duration of the launched process after which they terminate, hence the framework is daemon-less.

Consider a cluster with n compute nodes and c processor cores per node. Table 1 shows a comparison of times taken to spawn $n \times c$ processes on such a cluster. T_{conn} is the time taken to establish a connection to a node, T_{launch} is the time taken to spawn a single process and T_{nla} is the time taken to setup a NLA. We see that as the number of cores per node increases, the time taken to start the job decreases with the NLA approach. Since the dominant factor on most clusters is T_{conn} (around 5 ms on our testbed), the use of NLAs on multi-core systems keeps the spawn time practically constant for a fixed number of nodes irrespective of the number of cores per node.

Process Health. An important task of job launchers is to handle process termination. When a process fails, a job launcher must clean up other processes. Failure to do so would impact performance of future processes. Having a node

Table 1. Time Taken to Spawn Processes With and Without NLAs

With NLAs	Without NLAs
$n \times (T_{conn} + T_{nla}) + c \times T_{launch}$	$(n \times c) \times (T_{conn} + T_{launch})$

level agent allows ScELA to handle monitoring of process health in parallel. The NLAs monitor the health of local processes. When a failure is observed the NLA sends a PROCESS_FAIL notification message to the central launcher. The Launcher then sends a PROCESS_TERMINATE message to all other NLAs which terminate all processes. User signals such as SIGKILL are handled similarly.

2.2 NLA Interconnection Layer

Many programming models require some form of information exchange and synchronization between processes before they complete initialization. For instance, MPI processes may need to discover other processes on the same node to utilize efficient shared memory communication channels or processes may need a barrier synchronization before they can enter a subsequent phase of initialization. Having a connection between every process does not scale for a large number of processes as the number of connections required is $O(n^2)$. Other approaches have all processes connect to a central controller which coordinates information exchange and synchronization. However, when a large number of processes initiate connections to a central controller, it becomes a bottleneck. The resultant network congestion causes TCP SYN packets being dropped. Since SYN retransmission timeouts increase with every attempt on most TCP implementations [8], this introduces a large delay in the overall launch process. In addition, most operating systems limit the number of connections that can be kept open which makes a central controller unfeasible.

We have designed a communication layer over the NLAs to facilitate communication and synchronization between processes. Each NLA aggregates initialization information from all processes on the node. This aggregation limits the total number of network connections needed per entity (process, NLA or the Launcher) on the system. NLAs from different nodes form a hierarchical k-ary tree [9] for communication of information between processes across nodes. The hierarchical tree improves overall parallelism in communication. A k-ary tree allows ScELA to launch processes over an arbitrary number of nodes while also keeping the number of steps required for synchronization and other collective operations such as broadcast or gather at a minimum at $log_k(n)$ where n is the number of nodes. An example of a 3-ary tree of depth 3 is given in Figure 2.

The degree k of the k-ary tree determines the scalability and the performance of ScELA. An NLA in the hierarchical tree should be able to handle connection setup and communication from all processes on a node as well as the parent

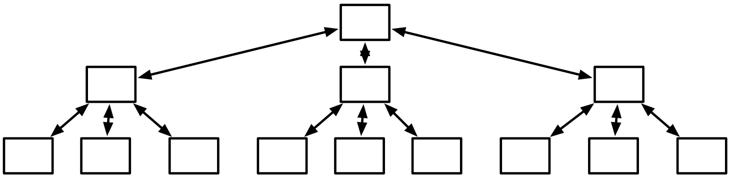


Fig. 2. Example 3-ary NLA Interconnection (with depth 3)

and children in the NLA tree. If the degree of the tree is too high, each NLA would have to process too many connections which would create further bottlenecks. If the degree is too low, the depth of the tree would result in too many communication hops.

We determine the degree k dynamically. If n is the number of nodes, we determine an ideal degree k such that the number of levels in the tree, $log_k(n)$, is as follows: $log_k(n) \leq MAX\_DEPTH$. If c is the number of cores per node and $c + k + 1 \leq MAX\_CONN$, then we select k as the degree. If not, we select $k = MAX\_CONN - 1 - c$. The parameter MAX_CONN is the number of connections that an entity can process in parallel without performance degradation. From our experiments (Section 4.2) we have determined that a process can handle up to 128 connections with acceptable performance degradation on current generation systems.

2.3 Communication Primitives

The characteristics of the information exchange between processes depends on the programming model as well as specific implementations. The communication pattern could be point-to-point, collective communication such as broadcast, reduce, or a protocol such as a global bulletin board. We have designed the following communication primitives over the NLA Interconnection Layer for use by the processes for efficient communication.

Point-to-point Communication Primitives: Some initialization protocols have processes communicating directly with each other. For such protocols, we have designed two sided point-to-point communication primitives – `NLA_Send` and `NLA_Recv`.

The data from a sender is forwarded to the receiver over the NLA tree. Each process is assigned a unique identifier. During the setup of the NLA Interconnection Layer, every NLA discovers the location of each process. A process is either on the same node as the NLA, or it can be found in specific lower branch of the NLA tree or higher up the NLA tree.

Collective Communication Primitives: In most programming models, all processes go through identical initialization phases with identical communication patterns. These communication protocols resemble MPI-style collective communication. To support such protocols, we have designed MPI-style collective communication primitives `NLA_Gather`, `NLA_Broadcast`, `NLA_AllGather`, `NLA_Scatter` and `NLA_AllToAll` over the NLA tree.

Bulletin Board Primitives: Some communication protocols have processes publish information about themselves on a global bulletin board and processes needing that information read it off the bulletin board. To support such protocols over ScELA we have designed two primitives – `NLA_Put` and `NLA_Get`.

`NLA_Put` publishes data to all NLAs up the tree up to the root. When a process needs to read data, it invokes the `NLA_Get` primitive. When data is not available at a NLA, it forwards the request to the parent NLA. When data is found at a higher level NLA, it is sent down the tree to the requesting NLA.

Synchronization Primitive: In some programming models, the information exchange phase consists of smaller sub-phases with synchronization of the processes at the end of each sub-phase. For instance, in MVAPICH, processes cannot initiate InfiniBand [10] channels until all processes have pre-posted receive buffers on the NIC.

We have designed a synchronization primitive – `NLA_Barrier` which provides barrier synchronization over the NLA tree. Processes are released from an invocation of `NLA_Barrier` primitive only when all other processes have invoked the primitive. The `NLA_Barrier` primitive can be used in conjunction with `NLA_Send` and `NLA_Recv` to design other forms of communication required by a specific communication protocol.

2.4 Hierarchical Cache

On multi-core nodes, with communication patterns such as the use of a bulletin board, many processes on a node may request the same information during initialization. To take advantage of such patterns, we have designed a NLA level cache for frequently accessed data. When a process posts information through `NLA_Put`, the data is sent up to the root of the NLA tree while also being cached at intermediate levels. When a process requests information through `NLA_Get`, the request is forwarded up the NLA tree until it is found at a NLA. The response gets cached at all intermediate levels of the tree. Hence subsequent processes requesting the same piece of information are served from a nearer cache. This reduces network traffic and improves the overall responsiveness of the information exchange.

Such a cache is advantageous even on non multi-core nodes or communication patterns without repeated access to common information because the caching mechanism propagates information down the NLA tree. Subsequent requests from other sub-branches of the tree may be served from an intermediate NLA and would not have to go up to the root. In Section 3.1 we describe an extension to the `PMI_Put` primitive that enables better utilization of the Hierarchical Cache.

2.5 Communication Protocols

As described in Section 2.3, the processes being launched may have their own protocol for communicating initialization information. We have designed the ScELA framework to be extensible so that various communication protocols can be developed over it by using the basic communication primitives provided. In Section 3 we describe two implementations of such protocols over the ScELA architecture.

3 Case Studies

In this section we describe implementations of two startup protocols over ScELA. We first describe an implementation of the Process Management Interface (PMI), an information exchange protocol used by popular MPI libraries such as MPICH2 and MVAPICH2 over the ScELA framework. In addition, we describe an implementation of another startup protocol – PMGR used by MPI libraries such as MVICH [11] and MVAPICH.

3.1 Designing the PMI Bulletin Board with ScELA

When MPI processes start up, they invoke `MPI_Init` to set up the parallel environment. This phase involves discovery of other processes in the parallel job and exchange of information. The PMI protocol defines a *bulletin board* mechanism for information exchange. Processes do a `PMI_Put` operation on a `(key, value)` pair to publish information followed by a `PMI_Commit` to make the published information visible to all other processes. When other processes need to read information, they perform a `PMI_Get` operation by specifying a `key`. The PMI protocol also defines a barrier synchronization primitive `PMI_Barrier`.

To implement the PMI bulletin board over the ScELA framework, we utilized the `NLA_Put` and `NLA_Get` primitives. A `PMI_Put` by a process invokes a corresponding `NLA_Put` to propagate information over the NLA tree. When a process does a `PMI_Get`, a corresponding `NLA_Get` is invoked to search for information in the Hierarchical Cache. Since the `PMI_Puts` are propagated immediately, we ignore `PMI_Commit` operations.

We have observed that with the PMI protocol, information reuse is high for some information. In such cases it is beneficial to populate the node level caches even before the first `PMI_Get` request. We have designed an extension to the `NLA_Put` primitive that propagates information to all NLAs in the tree so that all `NLA_Gets` can be served from a local cache. To reduce the number of `NLA_Puts` active in the tree, we aggregate puts from all processes on a node before propagating this information over the tree. When processes invoke `PMI_Barrier`, we invoke the `NLA_Barrier` primitive to synchronize processes. We evaluate our design against the current startup mechanism in MVAPICH2 in Section 4.1.

3.2 Designing PMGR (Collective Startup) with ScELA

The PMGR protocol defines MPI style collectives for communication of initialization data during `MPI_Init`. These operations also act as implicit synchronization between processes. The PMGR interface defines a set of collective operations – `PMGR_Gather`, `PMGR_Broadcast`, `PMGR_AlltoAll`, `PMGR_AllGather` and `PMGR_Scatter` and an explicit synchronization operation `PMGR_Barrier`.

In our implementation when a process invokes a PMGR primitive, it is directly translated to an invocation of the corresponding collective communication primitive designed over the NLA tree. We evaluate our design against the current startup mechanism in MVAPICH in Section 4.2.

4 Performance Evaluation

In this section we evaluate the two case studies described in Section 3. We evaluate our designs against the previous launching mechanisms in MVAPICH2 and MVAPICH respectively. Our testbed is a 64 node InfiniBand Linux cluster. Each node has dual 2.33 GHz Intel Xeon "Clovertown" quad-core processor for a total of 8 cores per node. The nodes have a Gigabit Ethernet adapter for management traffic such as job launching. We represent cluster size as $n \times c$, where n is the number of nodes and c is the number of cores per node used.

We have written an MPI microbenchmark to measure the time taken to launch MPI processes and time spent in `MPI_Init`, which represents the information exchange phase. For the purpose of these microbenchmark level tests, we disable all optional features that impact job initialization.

4.1 PMI over ScELA

In this section, we compare the performance of our implementation of PMI over ScELA (ScELA-PMI) against the default launch framework in MVAPICH2 (MVAPICH2-PMI). The default startup mechanism of MVAPICH2 utilizes a ring of daemons – `MPD` [12] on the target nodes. The launcher – `mpiexec` identifies target nodes and instructs the MPD ring to launch processes on them. PMI information exchange is done over the MPD ring. Figure 3 shows the time taken to establish the initial ring. We observe a linear increase which is not scalable over larger number of nodes. We have also observed that the MPD ring cannot be setup on larger sizes such as thousands of nodes. While a MPD ring can be reused for launching subsequent MPI jobs, most job schedulers elect to establish a separate ring as both target nodes and job sizes may be different. Figure 4 shows a comparison of the launch times for various system sizes. On ScELA-PMI, the spawn phase represents the time taken for the Launcher to setup

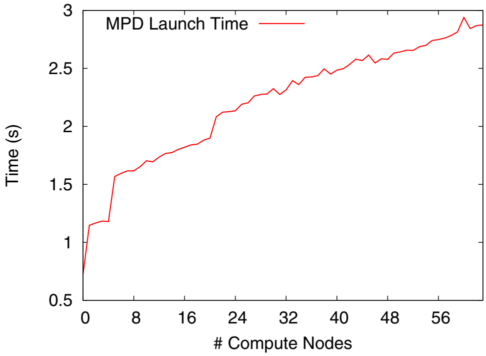
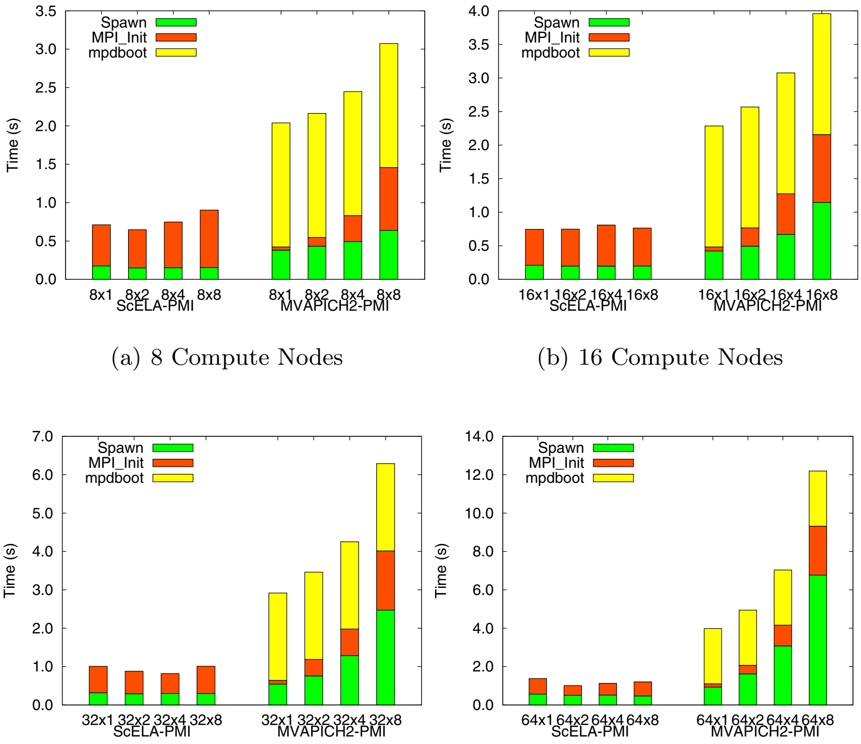


Fig. 3. Time to Setup MPD Ring with MVAPICH2



(a) 8 Compute Nodes

(b) 16 Compute Nodes

(c) 32 Compute Nodes

(d) 64 Compute Nodes

Fig. 4. Comparison of Startup Time on MVAPICH2

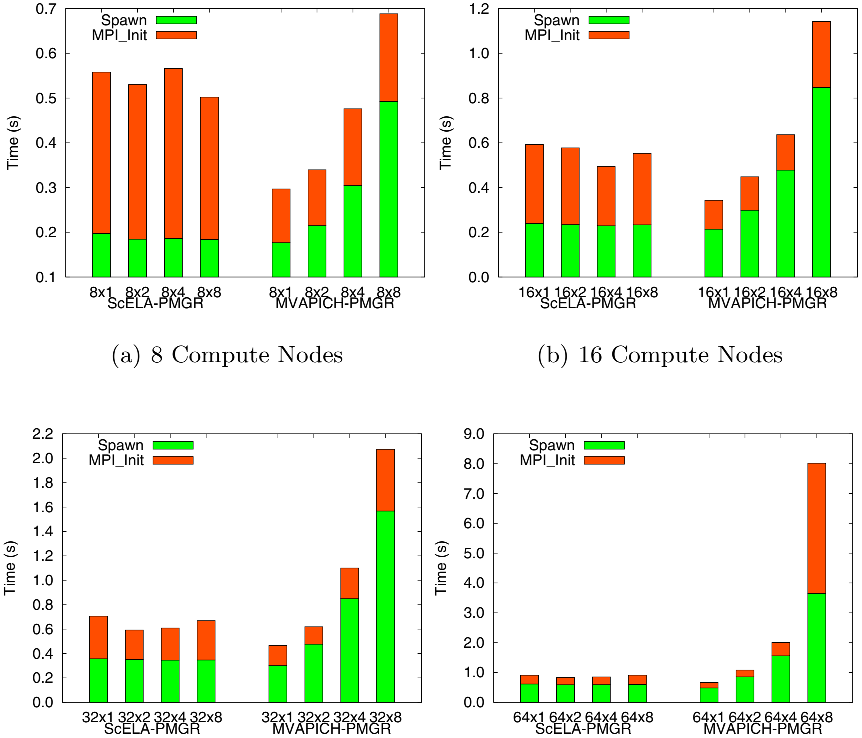
NLAs on the target nodes and for the NLAs to launch the MPI processes. The MPI_Init phase represents the time taken to establish the NLA Interconnection Layer and for PMI information exchange. On MVAPICH2-PMI the mpdboot phase represents the time taken to establish the ring of MPD daemons. The spawn phase represents time taken to launch MPI processes over the MPD ring and the MPI_Init phase represents the time taken for information exchange.

We observe that as we increase the number of processes per node, ScELA-PMI demonstrates better scalability. For a fixed node count, the duration of the spawn phase in ScELA-PMI is constant due to parallelism achieved through NLAs. In Figure 4(d) we see the spawn time for MVAPICH2-PMI increase from around $1s$ to $6.7s$ when the number of cores used per node is increased from 1 to 8 but ScELA-PMI is able to keep spawn time constant at around $0.5s$. At larger job sizes, for instance 512 processes on 64 nodes (64×8 in Figure 4(d)), we see an improvement in the MPI_Init phase from around $2.5s$ to $0.7s$ due to the better response times of communication over the NLA Interconnection Layer and due to reduced network communication due to NLA cache hits.

4.2 PMGR over ScELA

In this section we compare our design of PMGR over ScELA (ScELA-PMGR) against the default startup mechanism in MVAPICH (MVAPICH-PMGR). The default startup mechanism in MVAPICH has a central launcher that establishes a connection to target nodes and launches each process individually. On multi-core systems, this launcher needs several connections to each node. Also, each MPI process establishes a connection to the central controller which facilitates the PMGR information exchange. As the number of processes increase, this causes a flood of incoming connections at the central controller, which leads to delays due to serialization of handling these requests and network congestion. The number of MPI processes that can be handled simultaneously is also limited by resource constraints such as open file descriptor limits, which is typically 1024.

Figure 5 shows a comparison of the launch times. With ScELA-PMGR, the spawn phase represents the time taken to setup NLAs on the target nodes and for the NLAs to launch MPI processes on the node. The MPI_Init phase represents the time taken to setup the NLA Interconnection Layer and the PMGR information exchange between MPI processes. With MVAPICH-PMGR, the spawn



(a) 8 Compute Nodes

(b) 16 Compute Nodes

(c) 32 Compute Nodes

(d) 64 Compute Nodes

Fig. 5. Comparison of Startup Time on MVAPICH

phase represents the time taken for the central controller to launch each MPI process on target nodes. In the MPI_Init phase, the MPI processes establish connections to the central controller and exchange information over the PMGR protocol. We see that for a fixed node count, ScELA-PMGR takes constant time for the spawn phase as it benefits from having NLAs while the spawn phase with MVAPICH-PMGR grows with increase in number of processes per node. For instance in 5(d), we see that ScELA-PMGR is able to keep spawn time constant at 0.6*s*, but on MVAPICH-PMGR the spawn phase increases from 0.5*s* to 3.6 as we increase the number of cores used per node from 1 to 8. When the overall job size is small, the central controller in the MVAPICH startup mechanism is not inundated by a large number of connections. We see that the central controller is able to handle connections from up to 128 processes with little performance degradation in the MPI_Init phase. Hence the MVAPICH startup performs better at a small scale, but as the job sizes increase we observe larger delays in the MPI_Init phase. From figure 5(d) we see that for 512 processes (64×8), the MPI_Init phase takes 4.3*s* on MVAPICH-PMGR, but on ScELA-PMGR it takes around 0.3*s*. For 512 processes we see an improvement of 800% in the overall launch time.

Figure 6 shows a comparision of ScELA-PMGR and MVAPICH-PMGR on a large scale cluster – the TACC Ranger [13]. The TACC Ranger is an InfiniBand cluster with 3,936 nodes with four 2.0 GHz Quad-Core AMD "Barcelona" Opteron processors making a total of 16 processing cores per node. The Figure shows the runtime of a simple *hello world* MPI program that initializes the MPI environment and terminates immediately. In terms of number of processing cores, ScELA-PMGR scales up to at least three times more than MVAPICH-PMGR (based on MVAPICH version 0.9.9). On 10,240 cores, we observe that MVAPICH-PMGR takes around 185*s* while ScELA-PMGR takes around 25*s* which represents a speedup of more than 700%. We also see that MVAPICH-PMGR is unable to scale beyond 10,240 cores, while ScELA-PMGR is able to scale to at least 3 times that number.

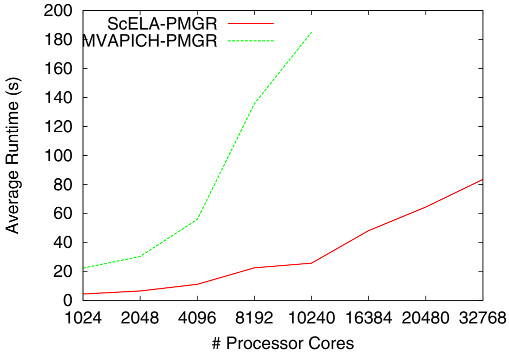


Fig. 6. Runtime of Hello World Program on a Large Scale Cluster (Courtesy TACC)

5 Related Work

The scalability and performance of job startup mechanisms in clusters have been studied in depth before. Yu, et. al. [14] have previously explored reducing the volume of data exchanged during initialization of MPI programs in InfiniBand clusters.

In our work, we have assumed availability of executable files on target nodes through network based storage as this is a common model on modern clusters. Brightwell, et. al. [15] have proposed a job-startup mechanism where network storage is not available.

SLURM [16] is a resource manager for Linux clusters that implements various interfaces such as PMI and PMGR for starting and monitoring parallel jobs. Unlike ScELA, SLURM has persistent daemons on all nodes through which it starts and monitors processes.

6 Conclusion and Future Work

Clusters continue to scale in core counts. Node counts are increasing significantly, but much of the growth in core counts is coming from multi-core clusters. In this paper we have demonstrated a scalable launching architecture that improves the launch performance on multi-core clusters by more than an order of magnitude than previous solutions. Although our case studies have been with two MPI libraries, we have presented an architecture extensible to any cluster launching requirements. For launching parallel jobs, we provide scalable and efficient communication primitives for job initialization.

With an implementation of our architecture, we have achieved a speedup of 700% in MPI job launch time on a very large scale cluster at $10,240$ processing cores by taking advantage of multi-core nodes. We have demonstrated scalability up to at least $32,768$ cores.

These solutions are being used by several large scale clusters running MVAPICH such as the TACC Ranger – currently the largest InfiniBand based computing system for open research.

In our solution, the Launcher launches Node Launch Agents serially. As node counts increase, this could be a potential bottleneck. However, this can be easily extended so that NLAs are launched hierarchically by other previously launched NLAs. We plan to explore this mechanism in the future.

With the recent demonstration of a 80 core processor by Intel, the number of cores per node on large scale clusters is projected to increase further. We can use more efficient communication channels such as UDP or shared memory for communication between processes and the NLA on a node so that the degree of the NLA tree can be decoupled from the number of cores on a node.

Software Distribution: Our implementation of PMGR over ScELA (ScELA-PMGR) is integrated with the 1.0 release of MVAPICH. The PMI implementation over ScELA (ScELA-PMI) will be available with the upcoming 1.2 release of MVAPICH2 [17].

Acknowledgment

We would like to thank Dr. Karl W. Schulz from TACC for helping us evaluate our designs on the TACC Ranger system. We would also like to thank Mr. Adam Moody from LLNL for providing the PMGR framework in MVAPICH.

References

1. TOP 500 Project: Top 500 Supercomputer Sites, `http://www.top500.org`
2. Sandia National Laboratories: Thunderbird Linux Cluster, `http://www.cs.sandia.gov/platforms/Thunderbird.html`
3. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (1994)
4. Network-Based Computing Laboratory: MVAPICH: MPI-1 over InfiniBand, `http://mvapich.cse.ohio-state.edu/overview/mvapich`
5. Argonne National Laboratory: MPICH2 : High-performance and Widely Portable MPI, `http://www.mcs.anl.gov/research/projects/mpich2/`
6. Huang, W., Santhanaraman, G., Jin, H.-W., Gao, Q., Panda, D.K.: Design of high performance mvapich2: Mpi2 over infiniband. In: Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2006) (2006)
7. Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E., Warren, K.: Introduction to upc and language specification. IDA Center for Computing Sciences (1999)
8. Shukla, A., Brecht, T.: Tcp connection management mechanisms for improving internet server performance. In: 1st IEEE Workshop on Hot Topics in Web Systems and Technologies, 2006. HOTWEB 2006, November 13-14, 2006, pp. 1–12 (2006)
9. Moody, A., Fernandez, J., Petrini, F., Panda, D.: Scalable NIC-based Reduction on Large-scale Clusters. In: Supercomputing, 2003 ACM/IEEE Conference (2003)
10. InfiniBand Trade Association: InfiniBand Architecture Specification, `http://www.infinibandta.com`
11. Lawrence Berkeley National Laboratory: MVICH: MPI for Virtual Interface Architecture (2001), `http://www.nersc.gov/research/FTG/mvich/index.html`
12. Butler, R., Gropp, W., Lusk, E.: Components and interfaces of a process management system for parallel programs. In: Parallel Computing (2001)
13. Texas Advanced Computing Center: HPC Systems, `http://www.tacc.utexas.edu/resources/hpcsystems/`
14. Yu, W., Wu, J., Panda, D.K.: Scalable startup of parallel programs over infiniband. In: Bougé, L., Prasanna, V.K. (eds.) HiPC 2004. LNCS, vol. 3296. Springer, Heidelberg (2004)
15. Brightwell, R., Fisk, L.: Scalable parallel application launch on cplant. In: Supercomputing, ACM/IEEE 2001 Conference, November 10-16 (2001)
16. Lawrence Livermore National Laboratory and Hewlett Packard and Bull and Linux NetworX: Simple Linux Utility for Resource Management, `https://computing.llnl.gov/linux/slurm/`
17. Network-based Computing Laboratory: (MVAPICH: MPI over InfiniBband and iWARP), `http://mvapich.cse.ohio-state.edu`

Parallel Information Theory Based Construction of Gene Regulatory Networks

Jaroslaw Zola[1], Maneesha Aluru[2], and Srinivas Aluru[1]

[1] Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50011, USA
[2] Department of Genetics, Cellular, and Developmental Biology
Iowa State University, Ames, IA 50011, USA
{zola,maluru,aluru}@iastate.edu

Abstract. We present a parallel method for construction of gene regulatory networks from large-scale gene expression data. Our method integrates mutual information, data processing inequality and statistical testing to detect significant dependencies between genes, and efficiently exploits parallelism inherent in such computations. We present a novel method to carry out permutation testing for assessing statistical significance while reducing its computational complexity by a factor of $\Theta(n^2)$, where n is the number of genes. Using both synthetic and known regulatory networks, we show that our method produces networks of quality similar to ARACNE, a widely used mutual information based method. We present a parallelization of the algorithm that, for the first time, allows construction of whole genome networks from thousands of microarray experiments using rigorous mutual information based methodology. We report the construction of a 15,147 gene network of the plant *Arabidopsis thaliana* from 2,996 microarray experiments on a 2,048-CPU Blue Gene/L in 45 minutes, thus addressing a grand challenge problem in the NSF *Arabidopsis* 2010 initiative.

Keywords: gene networks, mutual information, parallel computational biology, systems biology.

1 Introduction

Biological processes in every living organism are governed by complex interactions between thousands of genes and gene products. Modeling and understanding these interactions is essential to progress in many important research areas such as medicine, e.g. drug design, agriculture, e.g. pest-resistant crops, renewable energy, e.g. efficient production of biofuels, or more recently synthetic biology. Such interactions are typically modeled as gene regulatory networks [1], whose inference is one of the principal challenges in systems biology. A gene regulatory network is represented as a graph with vertices representing genes and edges representing regulatory interactions between genes. The functioning of a gene regulatory network in an organism determines the expression levels of various genes to help carry out a biological process. Network inference is the

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 336–349, 2008.

problem of predicting the underlying network from multiple observations of gene expressions (outputs of the network).

Rapid advances in high-throughput technologies, for instance microarrays [2] or the more recent short-read sequencing [3], enable simultaneous measurement of the expression levels of all genes in a given organism. The goal of network inference is to reconstruct the network from observations of gene expression data. The task is challenging due to several reasons: There are tens of thousands of genes in any complex organism, and in many cases it is impossible to find a reliable way to limit analysis to only a subset of them. Second, the number of available expression measurements falls significantly short of what is required by the underlying computational methods, with even the number of genes in the network significantly outnumbering the number of available measurements. At the same time, the computational complexity of the problem increases with increasing number of observations. Third, microarray measurement of expression data is inherently noisy and significantly influenced by many experimental-specific attributes, making it hard to derive meaningful conclusions. Finally, little is known about general regulatory mechanisms (e.g. post-transcriptional effects) and thus no satisfactory models of genetic regulation are available.

In spite of its difficulty the problem of reconstructing gene regulatory networks has been addressed by many researchers, and techniques like relevance networks [4,5], Gaussian graphical models [6,7] or Bayesian networks [8,9] have been widely adopted as possible approaches. While simple models like relevance networks or Gaussian graphical models perform well under the assumption of linear dependency between genes, they fail to detect non-linear types of interactions [5,10]. On the other hand more flexible models, like Bayesian networks, are limited by their exponential computational complexity, and require very large number of experiments even for networks of modest size [11]. As a result only simple models have been used to reconstruct genome-level regulatory networks [12]. In [13,14] Butte and Kohane and later Basso *et al.* suggested the use of mutual information [15] as a powerful method to detect any type of interactions between genes. Moreover, through experiments on synthetic data it has been shown that mutual information can be competitive to Bayesian networks [11,13] in terms of quality of generated solutions.

Microarray experiments are often conducted by individual research groups in small numbers (typically 6-10) to aid in specific biological investigations. However, such data is normally deposited in public microarray repositories (see for instance [16,17,18]), thus collectively providing a large number of experiments. Analyzing this data is even harder due to the significant variability in experimental protocols and conditions across laboratories, which render direct comparison of expression level data invalid. Sophisticated statistical methods are being developed for cross-platform comparisons, and having expression profile of every gene under different biological conditions enables scientists to ask more profound questions about nature of genetic interactions with the ultimate goal of accurate modeling of mechanisms that allow functioning of cells and organisms. The goal of our research was to develop parallel methods that can scale up to collectively analyzing all the microarray experiments available for an organism and build robust gene networks from this experimental data, whose collective economic value already runs into millions of dollars per organism for several organisms.

In this paper we present a method that combines mutual information, statistical testing and parallel processing to construct large whole genome regulatory networks. To reduce computational cost, we propose the B-spline mutual information estimator of Daub *et al.* [10] supported by permutation testing. We present a novel method to reduce the complexity of permutation testing by $\binom{n}{2}$, where n is the number of genes. Using synthetic and known regulatory networks we show that our method produces networks of quality similar to ARACNE [13], a widely used mutual information based network reconstruction software. We present a parallelization of the algorithm that allows construction of accurate whole genome networks from thousands of microarray experiments, a task which is beyond capabilities of sequential methods. We report the construction of a 15,147 gene network of *Arabidopsis thaliana* from 2,996 microarray experiments on a 2,048-CPU IBM Blue Gene/L in 45 minutes. This is the largest whole genome network published to date, both in terms of the number of genes and the number of microarray experiments analyzed to derive the network.

The remainder of this paper is organized as follows: In Section 2 we introduce the basic theory underlying mutual information and data processing inequality. In Section 3, we present a new way to carry out permutation testing whose cost is significantly reduced by amortizing over all pairs of genes. We propose a B-splines based mutual information estimator augmented with our permutation testing approach and demonstrate that it provides results of comparable quality to best currently known methods, which are limited in their scale. Our parallel method and its implementation are described in Section 4. In Section 5 we provide experimental results demonstrating quality and scalability, and report on the construction of the whole-genome network for the plant *Arabidopsis thaliana*. We conclude the paper in Section 6.

2 Concepts and Definitions

The problem of reconstructing a gene regulatory network can be formalized as follows: Consider a set of n genes $\{g_1, g_2, \ldots, g_n\}$, where for each gene a set of m expression measurements is given. We represent expression of gene i (g_i) as a random variable X_i with marginal probability p_{X_i} derived from some unknown joint probability characterizing the entire system. This random variable is described by observations $x_{i,1}, \ldots, x_{i,m}$, where $x_{i,j}$ corresponds to the expression level of g_i under condition j. We call the vector $\langle x_{i,1}, x_{i,2}, \ldots, x_{i,m} \rangle$ *profile* of g_i. Given a profile matrix $M_{n \times m}$, $M_{i,j} = x_{i,j}$ we want to find a network N that best explains the data in M.

2.1 Mutual Information and Data Processing Inequality

Intuitively, we assume that expression profiles of genes that are interacting will be correlated. To reconstruct regulatory network we first connect all pairs of genes that show statistically significant patterns of correlation, and then eliminate those edges that can be result of an indirect interaction (e.g. when two genes are coregulated by a third gene). This requires two principal components: a method to identify significant correlation, and a strategy to differentiate between direct and indirect interactions. Because interactions between genes are the result of sophisticated biochemical processes, and expression measurements are inherently noisy, we expect that many gene interactions will be exhibited in non-linear dependencies between their profiles. To address the above issues we follow results in [13] and we use information theoretic concepts of mutual information and data processing inequality [15].

Mutual information $\mathcal{I}(X_i; X_j)$ is arguably the best measure of correlation between two random variables X_i and X_j when the underlying relationship is non-linear [15]. It is defined based on entropy $\mathcal{H}(\cdot)$ in the following way:

$$\mathcal{I}(X_i; X_j) = \mathcal{H}(X_i) + \mathcal{H}(X_j) - \mathcal{H}(X_i, X_j), \tag{1}$$

where differential entropy $\mathcal{H}(X)$ of continuous variable X is given by:

$$\mathcal{H}(X) = -\int p_X(\xi) \log p_X(\xi) d\xi, \tag{2}$$

and p_X is a probability density function for X. In our case p_X is unknown and has to be estimated based on available gene expression observations, a problem which we discuss in the next section. Mutual information is symmetric and always non-negative. One way to interpret it is as a measure of information one variable provides about another. Therefore, $\mathcal{I}(X_i; X_j) = 0$ if and only if X_i and X_j are independent. Consequently, we will connect two genes in the reconstructed network if mutual information between their profiles is greater than a carefully chosen threshold. Note that estimating pairwise relationships between genes provides only partial knowledge about the underlying joint probability distribution. Unfortunately, analysis of higher order interactions would require much larger number of observations [19].

Having defined a correlation measure we are left with the task of identifying indirect interactions between genes. One way to solve this problem is to rely on conditional mutual information [15,19]. This approach, however, is not generally feasible because the number of available observations does not allow for accurate estimation of conditional mutual information. Therefore in [13] Basso *et al.* suggested to employ data processing inequality (DPI), another interesting property of mutual information. DPI states that if three random variables X_i, X_j, X_k form a Markov chain in that order (i.e., conditional distribution of X_k depends only on X_j and is independent of X_i), then $\mathcal{I}(X_i, X_k) \leq \mathcal{I}(X_i, X_j)$ and $\mathcal{I}(X_i, X_k) \leq \mathcal{I}(X_j, X_k)$. These inequalities can be used to discard indirect interactions: each time the pair (X_i, X_k) satisfies both inequalities the corresponding edge between g_i and g_k is removed from the network.

2.2 Mutual Information Estimation

To obtain mutual information between two random variables X_i and X_j, their marginal probability densities p_{X_i} and p_{X_j}, and their joint probability density

p_{X_i,X_j}, should be known or estimated (see equations (1) and (2)). Rather than make hard to justify assumptions about underlying probability distributions, we estimate them based on observed gene expression measurements. The estimates are then used to compute estimated mutual information. As the computed mutual information is an estimate of the true (but unknown and undeterminable) mutual information, a method to compute its statistical significance must be used for accurate assessment.

Several different techniques to estimate mutual information have been proposed (for example, see [20]). These methods differ on precision and computational complexity: simple histogram methods (used for example in [14]) are very fast but inaccurate, especially when the number of observations is small. On the other hand, kernel based methods, e.g. Gaussian kernel estimators utilized in [13], provide very good precision but are computationally demanding when the number of observations is large. Finally, to the best of our knowledge, all available mutual information estimators depend on some kind of parameter, like for example number of bins for the histogram estimator, or kernel bandwidth for kernel based techniques.

Consider random variable X_i and its m observations. The simplest way to estimate its probability density is to divide observations into fixed number of bins b and then count observations that fall into each bin. This technique is extremely fast but at the same time imprecise and sensitive to the selection of boundaries of bins [21]. To overcome this limitation Daub *et al.* have proposed to use B-splines as a smoothing criterion: each observation belongs to k bins simultaneously with weights given by B-spline functions up to order k. It is shown that the B-splines method produces results that are nearly perfectly correlated with the Gaussian kernel estimator, which is believed to be the most accurate procedure. In our own experiments, we found this to be true with a Pearson correlation coefficient of 0.9994 (1.0 denotes perfect correlation). Note that computational complexity of B-spline estimator is linear in m compared to $O(m^4)$ for the Gaussian kernel. Due to these reasons, we employ the B-splines technique.

3 A Novel Method for Permutation Testing

An important component to using $\mathcal{I}(X_i;X_j)$ to determine if there is significant dependency between genes g_i and g_j requires assessing the statistical significance of the quantity $\mathcal{I}(X_i;X_j)$ itself. This assessment is done through permutation testing. Let $\langle x_{i,1}, x_{i,2}, \ldots, x_{i,m}\rangle$ denote the sequence of m observations of g_i. Let $\pi(X_i) = \pi(\langle x_{i,1}, x_{i,2}, \ldots, x_{i,m}\rangle)$ denote a permutation of the vector of m observations. If there is significant dependency between X_i and X_j, it is expected that $\mathcal{I}(X_i;X_j)$ is significantly higher than $\mathcal{I}(\pi(X_i);X_j)$. The permutation testing method involves computing $\mathcal{I}(\pi(X_i);X_j)$ for all $m!$ permutations of $\langle x_{i,1}, x_{i,2}, \ldots, x_{i,m}\rangle$ and accepting the dependency between X_i and X_j to be statistically significant only if $\mathcal{I}(X_i;X_j) > \mathcal{I}(\pi(X_i);X_j)$ for at least a fraction $(1-\epsilon)$ of the $m!$ permutations tested, for some small constant $\epsilon > 0$. As testing all $m!$ permutation is computationally prohibitive for large m, a large sampling of the permutation space is considered adequate in practice.

Permutation testing is generally accepted as the benchmark for assessing statistical significance. However, all mutual information methods to assess significant dependencies between expression profiles avoid permutation testing due to its computational complexity. For example, ARACNE uses an equation derived from the large deviation theory that connects number of observations m and desired confidence level ϵ. Permutation tests for sample data sets of independent variables (sizes ranging from 100 to 360 observations) have been performed, and corresponding distribution of mutual information has been obtained. This distribution was then used to estimate parameters of the equation. Note that this is a one time effort to estimate the parameters using permutation tests on some sample data sets. The parameter fitted equation is what is recalled when ARACNE is normally used. This method has the obvious disadvantage of inaccurately estimating threshold values for input data sets with distribution of mutual information different than the one used to obtain parameters of the equation.

Ideally, permutation testing should be conducted for assessing the significance of each pair $\mathcal{I}(X_i; X_j)$ using a large number P of random permutations. This is clearly computationally prohibitive, increasing run time by a factor of P. In this paper, we present a novel method to collectively assess all pairwise mutual informations such that the permutation testing conducted for one pair can be used to assess the statistical significance of any other pair. Formally, we present a method to test $P/\binom{n}{2}$ random permutations per pair, while obtaining results equivalent to assessing each pair using P random permutations. This allows us to use only a few random permutations per pair, while obtaining statistically meaningful results. Our method works as follows:

It is well known that mutual information has the property of being invariant under homeomorphic transformations [15,22], that is:

$$\mathcal{I}(X_i; X_j) = \mathcal{I}(f(X_i); h(X_j)), \tag{3}$$

for any homeomorphisms f and h. Consider replacing the vector of observations for g_i, i.e., $\langle x_{i,1}, x_{i,2}, \ldots, x_{i,m} \rangle$ with the vector $\langle rank(x_{i,1}), rank(x_{i,2}), \ldots, rank(x_{i,m}) \rangle$, where $rank(x_{i,l})$ denotes the rank of $x_{i,l}$ in the set $\{x_{i,1}, x_{i,2}, \ldots, x_{i,m}\}$; i.e., we replace each gene expression value with its rank in the set of observed expression values for the gene. The transformation, which we term *rank transformation*, while not continuous, is considered a good approximation to homeomorphism [22]. Instead of computing mutual information of pairs of gene expression vectors directly, we equivalently compute the mutual information of their rank transformed counterparts. With this change, each gene expression vector is now a permutation of $\{1, 2, \ldots, m\}$. Therefore, a permutation $\pi(X_i)$ corresponds to some permutation of the observation vector of any other random variable X_j. Moreover, estimation of marginal probabilities required in equation (1) depends only on the number of observations, and thus can be computed collectively once for all expression profiles. More formally, consider applying permutation testing to a specific pair $\mathcal{I}(X_i; X_j)$ by computing $\mathcal{I}(\pi(X_i); X_j)$ for some randomly chosen permutation π. For any other pair $\mathcal{I}(X_k; X_l)$, $\exists \pi', \pi''$ such that $\pi(X_l) = \pi'(X_j)$ and $\pi(X_i) = \pi''(\pi'(X_k))$. Since π is a random permutation, so is

π'' and $\mathcal{I}(\pi(X_i); X_j)$ is a valid permutation test for assessing the statistical significance of $\mathcal{I}(X_k; X_l)$ as well. Thus, each permutation test is a valid test for all $\binom{n}{2}$ pairs of observations. Therefore, one can use a total of P permutation tests, instead of P permutation tests for each pair, reducing the work in permutation testing by a factor of $\Theta(n^2)$.

There are important side benefits to our approach with regards to both quality and computational efficiency: While permutation testing of a pair by itself is an agreed upon statistical technique, evaluating the significance of $\mathcal{I}(X_i; X_j)$ with respect to all $\mathcal{I}(X_i; X_k)$ (for $j \neq k$) is important to extract the more prominent interactions for a gene. This is naturally incorporated in our scheme as a fixed number of permutation tests are conducted on each pair, and then collectively used to assess the statistical significance of every pair. As for computational efficiency, if the observation vector for any X_i is a permutation of $1, 2, \ldots, m$, then $H(X_i)$ is the same for all X_i. Thus, equation (1) reduces to:

$$\mathcal{I}(X_i; X_j) = 2\mathcal{H}(\langle 1, 2, \ldots, m\rangle) - \mathcal{H}(X_i, X_j). \tag{4}$$

Hence, one can directly work with $\mathcal{H}(X_i, X_j)$. Because rank transformed data consists of equispaced observations, it also improves the performance of majority of mutual information estimators. Using our permutation testing technique combined with B-splines mutual information estimator, we have been able to obtain results of about the same quality as rigorous Gaussian kernel based programs such as ARACNE, with significantly faster run-times. These results are presented in Section 5.

4 Parallel Gene Network Reconstruction

Gene network reconstruction is both compute and memory intensive. Memory consumption arises from the size of input data if the number of observations is large, and also from the dense initial network generated in the first phase of the reconstruction algorithm. While network construction based on large number of observations is out of the scope of sequential computers, the computational costs are prohibitive as well. For example, in our experiments on an Intel Xeon 3GHz desktop, ARACNE took over seven days for constructing a network on 5,000 genes of *Arabidopsis thaliana* from 700 microarray experiments. Thus, parallel computers are essential to building robust whole genome networks from ever increasing repositories of gene expression data.

4.1 Our Parallel Method

Let p denote the number of processors. We use row-wise data distribution where each processor stores up to $\lceil \frac{n}{p} \rceil$ consecutive rows of the expression matrix M, and the same number of rows of the corresponding gene network adjacency matrix D. To start, each processor reads and parses its block of input data, and applies rank transformations to convert each gene expression vector to a permutation of ranks. The algorithm then proceeds in three phases: In the first phase, mutual information is computed for each of the $\binom{n}{2}$ pairs of genes, and q randomly chosen permutations per pair. Note that the total number of permutations used in the test $P = q \cdot \binom{n}{2}$, allowing a small constant value of q for large n. We employ the frequently utilized all-pairs parallel algorithm where the processors are treated as if connected in a ring, with $p-1$ shift permutations utilized such that expression vectors of all genes are streamed through each processor. This allows each processor to compute its portion of the adjacency matrix in p rounds. The q permutation tests per pair are also stored to determine threshold value in the second phase. The parallel run-time of the first phase is $O\left(\frac{qn^2}{p} + pt_s + nmt_w\right)$, where t_s is the network latency and $\frac{1}{t_w}$ is the network bandwidth.

In the second phase, the threshold value used as cutoff to assess significance of mutual information is computed, and edges below this threshold are discarded. The threshold is computed by finding the element with rank $(1-\epsilon)\cdot q \cdot \binom{n}{2}$ among the $q \cdot \binom{n}{2}$ mutual information values computed as part of permutation testing. While this can be computed using a parallel selection algorithm, note that ϵ is a small positive constant close to zero, and hence the threshold value is close to the largest value in the sorted order of the computed mutual information values. Let $r = \epsilon \cdot q \cdot \binom{n}{2}$. In our approach, each processor sorts its $O(\frac{qn^2}{p})$ values, and picks the r largest values. A parallel reduction operation is applied using the r largest values in each processor. The reduction operator performs linear merging of two samples of size r and retains the r largest elements. Once the r^{th} globally largest value is found, each processor eliminates from its local adjacency matrix edges that are below the threshold. The parallel run-time of this phase is $O\left(\frac{qn^2 \log n}{p} + \log p \cdot (r + t_s + rt_w)\right)$. Assuming $\epsilon < \frac{1}{p}$, linear scaling is expected.

The final phase of the algorithm is to apply data processing inequality. To decide if a given edge $D_{i,j}$ is the result of indirect interaction, complete information about rows i and j are needed. Because matrix D is stored row-wise, we need to stream row j to the processor responsible for row i. Moreover, because matrix D is symmetric it is sufficient to analyze its upper (or lower) triangular part. Once again, this is achieved in $p-1$ communication rounds, where in round i only processors with ranks $0, 1, \ldots, p-i$ participate in communication and processing. The parallel run-time of this phase is $O\left(\frac{n^3}{p} + pt_s + n^2 t_w\right)$. Note that while this worst case analysis indicates this to be the most compute intensive phase of the algorithm, it is not so. This is because DPI needs to be applied only to current existing edges in the network, and the network is expected to be significantly sparse. This observation is borne by experimental results which confirm that the run-time of the algorithm is dominated by the first phase of dense matrix computation.

4.2 Parallel Implementation

We developed a parallel gene regulatory network inference program implementing the methodology described in this paper. The program, termed *TINGe* for

Tool for Inferring Networks of Genes, is developed in C++ and MPI. While omitted due to brevity and scope issues, the program uses sophisticated statistical normalization techniques to make valid inferences across a large number of microarray experiments prior to execution of the presented methodology. To our knowledge, this is the first parallel gene network construction software, and it can scale to whole genome networks and the largest collections of microarray data collections presently available.

Apart from an implementation of the algorithms presented, we carried out a number of low level optimizations, and architecture specific optimizations targeted to our experimental platform, the Blue Gene/L. The mutual information estimator of Daub *et al.* consists of three main elements: a procedure to compute values of B-spline function, a joint probability calculation, and entropy calculation. We implemented B-spline functions based on the de Boor algorithm [23]. With the help of the PAPI profiler, we optimized main loops in the probability computations to efficiently exploit fused multiply-add instruction of Blue Gene/L PPC440 CPU. Finally, our implementation relies on highly optimized implementations of the *log* function provided by vendor's math kernel library (e.g. IBM's MASS library for Blue Gene). Because of the large size of the input data sets we tested (about 2GB for the largest one), we used collective MPI-IO operations which we found to be the most efficient on the Blue Gene/L system.

5 Experimental Results

To validate our approach and its implementation, we present experimental results that fall into three categories: First, we compare the accuracy and efficiency of our method with ARACNE [13], the best currently available program that uses mutual information and DPI to reconstruct gene regulatory networks. We also compare ARACNE with a sequential implementation of our approach, to show that the presented permutation testing with B-splines approach enables faster network construction even on a sequential basis. We then present a scalability analysis of our parallel implementation by testing on varying numbers of processors. Finally, we report on the construction of the whole genome network of the model plant *Arabidopsis thaliana* from 2,996 microarray experiments, which is the largest whole genome network reported to date.

5.1 Quality and Run-Time Assessment

We assessed the accuracy of our method using both synthetic and biological networks. It is common practice to validate network inference algorithms using the SynTReN package [24], a widely used software developed with the express purpose of providing common benchmark data sets with desired sizes (number of genes and experiments) and known underlying networks. We generated two synthetic regulatory networks using SynTReN, each consisting of 100 genes, but differing in the number of expression profiles (500 and 900, respectively). We used ARACNE and TINGe to reconstruct each network from expression data

Table 1. Comparison of ARACNE and TINGe using a 100 gene synthetic network with two different expression profiles

	$m = 500$		$m = 900$	
	ARACNE	TINGe	ARACNE	TINGe
Time (in sec)	90	6	400	10
Specificity	0.95	0.95	0.95	0.95
Sensitivity	0.59	0.60	0.61	0.61
Precision	0.25	0.26	0.25	0.25
TP	82	85	85	86
TN	4566	4562	4555	4550
FP	244	248	255	260
FN	58	55	55	54

and compared the generated networks with the known true network to check correctness of the predicted edges. Table 1 summarizes results obtained on a Pentium D 3GHz computer with 4GB of RAM running 64-bit Red Hat Linux OS (both codes have been compiled with the g++ 4.1.2 compiler). Here we use TP, TN, FP and FN to denote number of true positives (edges correctly predicted), number of true negatives (edges correctly avoided), number of false positives (edges incorrectly predicted) and number false negatives (incorrectly avoided edges), respectively. We assess the performance of both softwares using standard measures: i) Specificity – fraction of missing edges correctly classified ($\frac{TN}{TN+FP}$), ii) Sensitivity – fraction of correct edges predicted ($\frac{TP}{TP+FN}$), and iii) Precision – fraction of predicted edges that are correct ($\frac{TP}{TP+FP}$).

As can be seen from Table 1, TINGe outperforms ARACNE in execution times while it preserves high specificity and sensitivity. Note that as expected B-spline estimator scales linearly with the size of expression profile, while the overhead due to permutation testing does not offset resulting gain in efficiency. This demonstrates that our proposed approach of B-splines augmented with permutation testing can generate high quality networks with lower computational cost.

We also tested both programs on an extracted sample regulatory subnetwork of *Arabidopsis thaliana* from the AtRegNet database [25] (see Fig. 1). This network consists of 56 genes and 60 edges, and provides information about interactions that have been confirmed through biological experiments. We used expression profile of 2,996 microarrays (see Section 5.3 for explanation of how the data has been prepared) to reconstruct this network using both ARACNE and TINGe. At this point we should explain that verifying obtained results with simple criteria such as those used for synthetic data are hardly insightful. In particular, we have to be very mindful when interpreting false positive predictions. Very often such predictions correspond to indirect interactions that cannot be rejected without carefully designed and targeted biological experiments. Moreover, predictions of this type can still be highly valuable for biologists, especially when little is known about the investigated network. With these caveats in mind, TINGe provided 18 true positive predictions with 40 false negatives and 121 false positive predictions. ARACNE generated 11 true positives while rendering

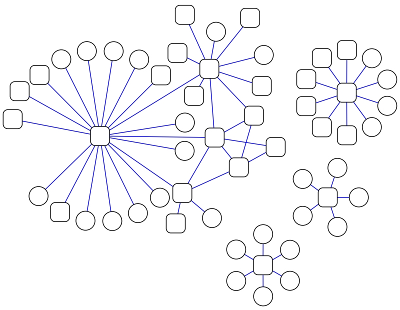


Fig. 1. Topology of a known gene regulatory network of *Arabidopsis thaliana* obtained from the AtRegNet database. Rectangular nodes denote transcription factors.

47 false negatives and 134 false positives. The time required to reconstruct this network was 10 seconds for TINGe, and 2,580 seconds for ARACNE.

5.2 Parallel Performance

To assess scalability of our approach, we performed a series of tests on our 2,048-CPU Blue Gene/L system using two data sets containing 424 and 2,996 microarray experiments, and 2,048 and 4,096 genes. Run-time for network reconstruction was measured as a function of the number of processors. The results are summarized in Table 2 and Fig. 2.

Table 2. TINGe runtime in seconds for different number of genes n, and different number of expression observations m

	$m = 424$		$m = 2996$	
# CPU	$n = 2048$	$n = 4096$	$n = 2048$	$n = 4096$
32	293	1173	2688	10731
64	151	602	1385	5524
128	77	306	704	2798
256	39	155	360	1421
512	21	80	188	723
1024	12	42	103	377

As can be seen, TINGe maintains linear scalability up to 1,024 CPUs. In each case the total execution time is dominated by pairwise mutual information computations, with second and third phases of the algorithm accounting for less than 5 seconds in the worst case. While the third stage of computing indirect edges is $O(n^3)$ in the worst-case, the actual complexity is $O(en)$, where e is the number of predicted edges (TP+FP). As expected, $e \ll n^2$ and the overall computation is dominated by the first phase with total work of $\Theta(n^2)$. While this

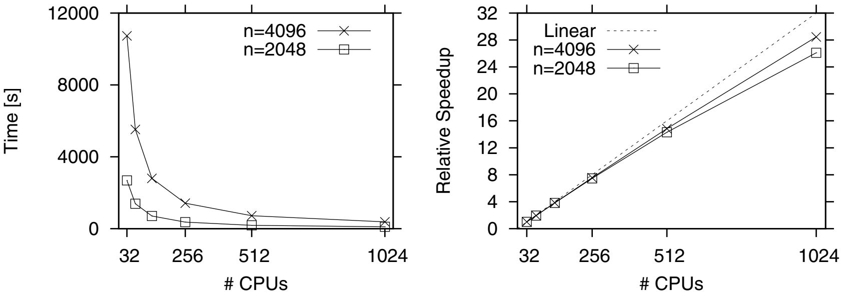


Fig. 2. Execution time (left) and relative speedup (right) as a function of number of CPUs for the data sets with 2,996 observations

may suggest that second and third phase could be handled sequentially, it should be kept in mind that this is infeasible for whole-genome scale problems due to memory constraints. As expected, the parallel run-time grows quadratically with the number of genes and linearly with the number of experiments.

5.3 Reconstructing Gene Regulatory Network of *Arabidopsis thaliana*

The main motivation behind this work is to enable, for the first time, genome-level reconstruction of gene regulatory networks based on robust and sensitive criterion of mutual information applied to large number of microarray experiments. The organism of particular interest to us is the model plant *Arabidopsis thaliana*. Understanding the functional roles of all of its genes and inferring the whole genome network of this organism are the primary goals of the 10-year U.S. National Science Foundation *Arabidopsis* 2010 initiative, launched in year 2000.

To reconstruct the gene regulatory network of *Arabidopsis*, we obtained a set of Affymetrix ATH1 microarrays from the main *Arabidopsis* repository at NASC [18]. This data, so called "Super Bulk Gene Download", is preliminarily processed using Affymetrix MAS-5.0 protocol [26]. Because the data is aggregation of experiments performed by different laboratories around the world, it is necessary to normalize the data to remove variability due to non-biological sources. This process is itself quite difficult and challenging [2,27]. In our case, we first removed experiments with missing values, which suggest errors in experiment preparation, and subjected the remaining set of profile data to quantile normalization. The latter step should guarantee that expression measurements are comparable across different experiments. From the resulting profile matrix we removed genes for which expression measurements did not cover a wide enough range of expression. This is because uniformly expressed genes are not informative and cannot be accurately correlated with others. At the end of the above process we obtained a profile matrix consisting of 15,147 genes and 2,996 microarray experiments. To our knowledge, it is the largest data set used to date for reconstructing a gene regulatory network. TINGe constructed a whole-genome network on this data on a 2,048-CPU Blue Gene/L in 45 minutes, where IO operations took 4 minutes, finding threshold value required 1 second and application of DPI ran in 16 seconds. Preliminary analysis of the network for both validation and discovery purposes has yielded significant results, and further research is ongoing.

6 Conclusions

In this paper, we presented the first parallel method to construct gene regulatory networks that can scale to whole genome networks and large number of cumulative microarray experiments conducted worldwide. We present an approach to significantly reduce the cost of permutation testing, while obtaining results equivalent to a much larger sampling of the permutation space. Using a combination of these techniques, we reported on building the largest whole-genome plant network known to date. This network is being used to understand several biological pathways and has already met with some early success in identifying unknown genes in partially characterized biological pathways.

The TINGe software is freely available together with documentation at the following URL: http://www.ece.iastate.edu/∼zola/tinge.

Acknowledgments

We wish to thank Daniel Nettleton for discussions regarding the validity of the permutation testing approach. This research is supported in part by the National Science Foundation under CCF–0811804.

References

1. Zhu, X., Gerstein, M., Snyder, M.: Getting connected: analysis and principles of biological networks. Genes & development 21(9), 1010–1024 (2007)
2. The chipping forecast II. Special Supplement. Nature Genetics (2002)
3. Torres, T., Metta, M., Ottenwalder, B., et al.: Gene expression profiling by massively parallel sequencing. Genome research 18(1), 172–177 (2008)
4. Butte, A., Kohane, I.: Unsupervised knowledge discovery in medical databases using relevance networks. In: Proc. of American Medical Informatics Association Symposium, pp. 711–715 (1999)
5. D'haeseleer, P., Wen, X., Fuhrman, S., et al.: Mining the gene expression matrix: Inferring gene relationships from large scale gene expression data. In: Information Processing in Cells and Tissues (1998)
6. de la Fuente, A., Bing, N., Hoeschele, I., et al.: Discovery of meaningful associations in genomic data using partial correlation coefficients. Bioinformatics 20(18), 3565–3574 (2004)
7. Schafer, J., Strimmer, K.: An empirical Bayes approach to inferring large-scale gene association networks. Bioinformatics 21(6), 754–764 (2005)
8. Friedman, N., Linial, M., Nachman, I., et al.: Using Bayesian networks to analyze expression data. Journal of Computational Biology 7, 601–620 (2000)

9. Yu, H., Smith, A., Wang, P., et al.: Using Bayesian network inference algorithms to recover molecular genetic regulatory networks. In: Proc. of International Conference on Systems Biology (2002)
10. Daub, C., Steuer, R., Selbig, J., et al.: Estimating mutual information using B-spline functions – an improved similarity measure for analysing gene expression data. BMC Bioinformatics 5, 118 (2004)
11. Hartemink, A.: Reverse engineering gene regulatory networks. Nature Biotechnology 23(5), 554–555 (2005)
12. Ma, S., Gong, Q., Bohnert, H.: An Arabidopsis gene network based on the graphical Gaussian model. Genome research 17(11), 1614–1625 (2007)
13. Basso, K., Margolin, A., Stolovitzky, G., et al.: Reverse engineering of regulatory networks in human B cells. Nature Genetics 37(4), 382–390 (2005)
14. Butte, A., Kohane, I.: Mutual information relevance networks: functional genomic clustering using pairwise entropy measurements. In: Pacific Symposium on Biocomputing, pp. 418–429 (2000)
15. Cover, T., Thomas, J.: Elements of Information Theory, 2nd edn. Wiley, Chichester (2006)
16. EMBL-EBI ArrayExpress (last visited) (2008), `http://www.ebi.ac.uk/microarray-as/aer/`
17. NCBI Gene Expression Omnibus (last visited) (2008), `http://www.ncbi.nlm.nih.gov/geo/`
18. NASC European Arabidopsis Stock Centre (last visited) (2008), `http://www.arabidopsis.info/`
19. Schneidman, E., Still, S., Berry, M., et al.: Network information and connected correlations. Physical review letters 91(23), 238701 (2003)
20. Khan, S., Bandyopadhyay, S., Ganguly, A., et al.: Relative performance of mutual information estimation methods for quantifying the dependence among short and noisy data. Physical review. E 76(2 Pt 2), 026209 (2007)
21. Moon, Y., Rajagopalan, B., Lall, U.: Estimation of mutual information using kernel density estimators. Physical review. E 52(3), 2318–2321 (1995)
22. Kraskov, A., Stogbauer, H., Grassberger, P.: Estimating mutual information. Physical review. E 69(6 Pt 2), 066138 (2004)
23. De Boor, C.: A practical guide to splines. Springer, Heidelberg (1978)
24. Van den Bulcke, T., Van Leemput, K., Naudts, B., et al.: SynTReN: a generator of synthetic gene expression data for design and analysis of structure learning algorithms. BMC Bioinformatics 7, 43 (2006)
25. Palaniswamy, S., James, S., Sun, H., et al.: AGRIS and AtRegNet. A platform to link cis-regulatory elements and transcription factors into regulatory networks. Plant physiology 140(3), 818–829 (2006)
26. Statistical algorithms description document (last visited) (2008), `http://www.affymetrix.com/`
27. Irizarry, R., Warren, D., Spencer, F., et al.: Multiple-laboratory comparison of microarray platforms. Nature Methods 2, 345–350 (2005)

Communication Analysis of Parallel 3D FFT for Flat Cartesian Meshes on Large Blue Gene Systems*

Anthony Chan[1], Pavan Balaji[2], William Gropp[3], and Rajeev Thakur[2]

[1] ASCI FLASH Center, University of Chicago
chan@mcs.anl.gov
[2] Math. and Comp. Sci. Division, Argonne National Laboratory
{balaji,thakur}@mcs.anl.gov
[3] Dept. of Computer Science, University of Illinois, Urbana-Champaign
wgropp@illinois.edu

Abstract. Parallel 3D FFT is a commonly used numerical method in scientific computing. P3DFFT is a recently proposed implementation of parallel 3D FFT that is designed to allow scalability to massively large systems such as Blue Gene. While there has been recent work that demonstrates such scalability on regular cartesian meshes (equal length in each dimension), its performance and scalability for flat cartesian meshes (much smaller length in one dimension) is still a concern. In this paper, we perform studies on a 16-rack (16384-node) Blue Gene/L system that demonstrates that a combination of the network topology and the communication pattern of P3DFFT can result in early network saturation and consequently performance loss. We also show that remapping processes on nodes and rotating the mesh by taking the communication properties of P3DFFT into consideration, can help alleviate this problem and improve performance by up to 48% in some special cases.

1 Introduction

Fast Fourier Transform (FFT) has been one of the most popular and widely used numerical methods in many areas of scientific computing, including digital speech and signal processing, solving partial differential equations, molecular dynamics [3], many-body simulations and monte carlo simulations [1,2,14]. Given its importance, there have been a large number of libraries that provide different implementations of FFT (both sequential and parallel) aimed at achieving high-performance in various environments. FFTW [15], IBM PESSL [13], and

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. We also acknowledge IBM for allowing us to use their BG-Watson system for our experiments. Finally, we thank Joerg Schumacher for providing us his test code that allowed us to understand the scalability issues with P3DFFT on flat cartesian meshes.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 350–364, 2008.

the Intel Math Kernel Library (MKL) [9] are a few examples of such implementations. P3DFFT [6] is a recently proposed parallel implementation of 3D FFT that is designed to allow scalability for large problem sizes on massively large systems such as Blue Gene (BG) [16]. It aims at achieving such scalability by limiting communication to processes in small local sub-communicators instead of communicating with all processes in the system.

While there has been previous work that demonstrates the scalability of P3DFFT for regular 3D cartesian meshes, where all dimensions of the mesh are of equal length [7], its inability to achieve similar scalability for flat 3D cartesian meshes, where one dimension is much smaller than the other two, is a known problem [18]. Flat 3D cartesian meshes are a good tool in studying quasi-2D systems that occur during the transition of 3D systems to 2D systems (e.g., in superconducting condensate [17], Quantum-Hall effect [20] and turbulence theory in geophysical studies [19]). Thus, such loss of scalability can be a serious problem that needs to be addressed.

In this paper, we analyze the performance of P3DFFT for flat 3D cartesian meshes on a large 16-rack (16384-node) Blue Gene/L (BG/L) system. Specifically, we perform detailed characterization of the communication pattern used by P3DFFT and its behavior on the BG network topology. We observe that a combination of the network topology and the communication pattern of P3DFFT can result in parts of the communication to over-saturate the network, while other parts under-utilize the network. This causes overall loss of performance on large-scale systems. We also show that carefully remapping processes on nodes and rotating the mesh by taking the communication properties of P3DFFT into consideration can help alleviate this problem. Our experimental results demonstrate up to 48% improvement in performance in some cases.

2 Overview of Parallel 3D FFT Techniques

FFT [8] is an efficient algorithm to compute the Discrete Fourier Transform (DFT) and its inverse. Fourier transform consists of a forward and a backward transform. The forward operation transforms a function *f(x)* in real space X to a function *F(k)* in Fourier space K. The backward transform does the reverse operation that transforms *F(k)* in Fourier space K to *f(x)* in real space X. In this section, we will mainly discuss the forward fourier transform, but the backward fourier transform can be similarly performed by reversing the steps in the forward transform.

A typical 3D forward fourier transform for a real-space function *f(x,y,z)* can be expressed as follows:

$$f(k_x, k_y, k_z) = \sum_z \left[\sum_y \left[\sum_x f(x, y, z) \cdot e^{ik_x x} \right] e^{ik_y y} \right] e^{ik_z z} \tag{1}$$

The goal here is to perform a 1D fourier transform on each of the three dimensions of the 3D data mesh, distributed over P processes. There are two

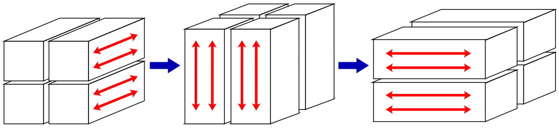


Fig. 1. 2D Decomposition: 1D FFT in each dimension followed by a transpose

basic approaches for doing this [11], distributed FFT and transpose-based FFT. Distributed FFT relies on a parallel implementation of 1D-FFT, with each process communicating the necessary data with other processes. Transpose-based FFT, on the other hand, relies on a sequential version of 1D-FFT that performs the transform on one dimension at a time, and transposing the data grid when needed. There are two different transpose-based FFT strategies for 3D meshes, which differ in their data decomposition pattern. Let us consider a data grid of size: $n_x \times n_y \times n_z$.

- *1D Decomposition:* In the 1D data decomposition technique, the data grid is divided across P processes such that each process gets a 2D slab of the grid (size of $n_x \cdot n_y \cdot n_z / P$). Each process carries out a typical sequential 2D-FFT on its local slab, and thus does not require any communication during this operation. Once the 2D-FFT has completed, it transposes the mesh using an `MPI_Alltoallv()` operation. This allows it to receive data corresponding to the third dimension, on which a 1D-FFT is applied. Thus, only one global transpose is used in this technique. However, the drawback is that it only scales `max(`n_x`,`n_y`,`n_z`)` number of processes.
- *2D Decomposition:* In the 2D data decomposition technique (shown in Figure 1), one face (2D) of the mesh is divided over $P = P_{row} \times P_{col}$ processes, so each process contains a column (pencil) of the data mesh of size $n_x \times (n_y/P_{row}) \times (n_z/P_{col})$. Each process first performs a 1D-FFT along the length of the column (say x-axis). Then it does a transpose on the remaining two axes (y- and z-axis) and performs 1D-FFT on the y-axis. Finally, it performs a transpose on the y- and z-axes and performs a 1D-FFT on the z-axis. Two transposes are performed altogether. P3DFFT uses this strategy as it can theoretically scale up to $n_x \cdot n_y \cdot n_z$ `/ min(`n_x`,`n_y`,`n_z`)` processes.

Neither of the transpose based FFT techniques allows for easy overlap of communication and computation as the transpose where the communication takes places has to be finished before the local 1D-FFT can be carried out.

3 Related Work

A number of implementations of Parallel 3D FFT exist. FFTW [4] has been a popular implementation of parallel 3D FFT. While there has been prior literature [10] that identified issues with its performance and improved its scalability to some extent, FFTW itself relies on 1D decomposition (described in Section 2) which allows it to only scale up to a theoretical limit of `max(`n_x`,`n_y`,`n_z`)` number of processes. That is, with a problem size of 4096^3, FFTW cannot use more than 4096 processors. Thus, it is not ideal to use on massively parallel systems such as BG which support hundreds of thousands of processors. P3DFFT has recently been proposed to deal with such scalability issues and allow 3D FFT to be effectively used on such systems.

Like P3DFFT, IBM recently proposed an alternate implementation of Parallel 3D FFT, specifically for their BG system, known as BGL3DFFT [12]. However, BGL3DFFT has several limitations. First, it is a closed source implementation that restricts much utility for open research. Second, owing to its lack of Fortran support, it has not gained too much popularity in mainstream scientific computing. Third, while there is published literature that shows its scalability for small 3D grids (up to $128 \times 128 \times 128$) [12], there is no evidence of its scalability for larger problem sizes. Keeping the drawbacks of BGL3DFFT aside, we believe that the problems in handling flat cartesian problems exist in the BGL3DFFT implementation as well, and that our observations are relevant there too.

There is also previous literature that shows that P3DFFT scales reasonably well with large regular cubical 3D meshes [7]. However, recently, Joerg Schumacher pointed out the importance of 3D-FFT on flat cartesian meshes where $n_x = n_y > n_z$ in his crossover study from 3D to quasi-2D turbulence systems [18] and found that 3D-FFT on flat cartesian meshes does not scale as well as regular cartesian meshes. Our paper uses Joerg's study as a motivation to understand the scalability issues of P3DFFT for flat cartesian meshes.

4 Communication Overheads in P3DFFT

In this section, we first present relevant details about the BG network in Section 4.1. We next present the communication characteristics of P3DFFT in Section 4.3 and an analysis of network saturation caused by such communication in Section 4.2.

4.1 BG/L Network Overview

BG/L has five different networks [5]. Two of them (1G Ethernet and 100M Ethernet with JTAG interface) are used for file I/O and system management while the other three (3-D Torus Network, Global Collective Network and Global Interrupt Network) are used for MPI communication. The 3-D torus network is used for point-to-point MPI and multicast operations and connects all compute nodes to form a 3-D torus; thus, each node has six neighbors. Each link provides a bandwidth of 175 MB/s per direction for a total of 1.05 GB/s bidirectional bandwidth per node.

4.2 Analyzing Network Saturation Behavior in P3DFFT on BG/L

As described earlier, unlike regular clusters that use switched network fabrics, the Blue Gene family of supercomputers relies on a torus network for interconnectivity. Thus, each node is directly connected with only six other nodes. To

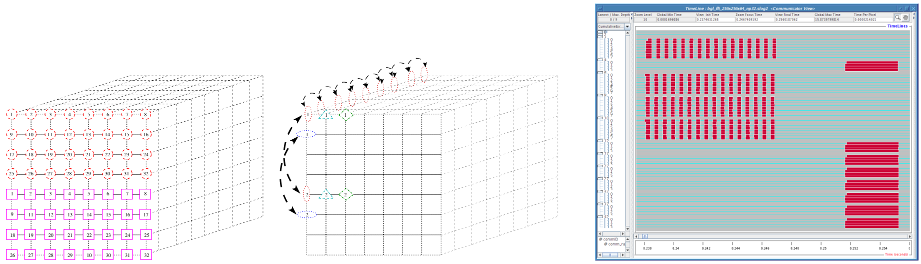


Fig. 2. (a) Mapping the row and column communicator processes in a 2D process grid to a 3D torus; (b) Jumpshot's communicator view on P3DFFT's communication

reach any other node, the message has to traverse more than one link; this leads to network link sharing by multiple messages, leading to network saturation.

Since P3DFFT does not directly perform communication with all processes in the system, but rather communicates only with processes in its row and column sub-communicators, the network saturation behavior is tricky. In Figure 2(a), we show the mapping of the processes in the row and column sub-communicators to the physical torus on BG/L. This example considers a system size of 512 processes, with the row sub-communicator containing 32 processes and the column sub-communicator containing 16 processes, i.e., a 32×16 process grid. Thus, the first row would have processes 1 to 32, the second row would have processes 33 to 64 and so on.

Note that the size of different dimensions in the BG/L torus is fixed based on the available allocation. In this case, we pick a torus topology of $8 \times 8 \times 8$. Therefore, the first row of processes in the process grid (1 to 32) map to the first four physical rows on the BG/L torus (shown as red circles in Figure 2(a)). Similarly, the second row of processes in the process grid (33 to 64) map to the next four physical rows (shown as pink rectangles). It is to be noted that all processes in the row communicator are always allocated adjacent to each other. That is, any communication within the row sub-communicator will not require the message to go outside these four rows.

The mapping of the processes corresponding to the column communicator is, unfortunately, more complicated than the row communicator. Processes corresponding to the first column are 1, 33, 65, 97, etc. These processes are not all topologically adjacent. In other words, as shown in Figure 2, messages traversing the non-adjacent portions of the column communicator have to pass through more links, oftentimes contending with messages from other communicators, and can thus saturate the network significantly faster as compared to the row communicator.

4.3 Communication Characterization of P3DFFT

Consider a 3D data grid of size $N = n_x \times n_y \times n_z$ which needs to be solved on a system with P processes. P3DFFT decomposes the 3D grid into a 2D processor mesh of size $P_{row} \times P_{col}$, where $P_{row} \times P_{col} = P$. It splits the 2D processor mesh into two orthogonal sub-communicators—one in each dimension. Thus, each process will be a part of a *row* and a *column* sub-communicator. As shown in Figure 2(b), the first global transpose of the forward 3D-FFT consists of n_z/P_{col} iterations of `MPI_Alltoallv` over the row sub-communicator (the short red states), with the message count per process-pair being m_{row} defined in Equation 2. The total message count per process for the first transpose becomes $n_x \cdot n_y \cdot n_z/(P_{row} \cdot P_{col})$.

$$m_{row} = \frac{n_x \cdot n_y}{P_{row}^2} = \frac{N}{n_z \cdot P_{row}^2} \tag{2}$$

The second global transpose consists of one single iteration of `MPI_Alltoallv` over the column communicator (the long red states in Figure 2(b)), with message count per process being $n_x \cdot n_y \cdot n_z/(P_{row} \cdot P_{col})$, which is the same as the first transpose. The corresponding message count per process-pair is m_{col}, where

$$m_{col} = \frac{n_x \cdot n_y \cdot n_z}{P_{row} \cdot P_{col} \cdot P_{col}} = \frac{N \cdot P_{row}}{P^2} \tag{3}$$

The total communication cost for the two global transposes becomes:

$$\begin{aligned} T(n_z, P_{row}) &= \frac{n_z}{P_{col}} \cdot T_{row}(m_{row}) + T_{col}(m_{col}) \\ &= \frac{n_z \cdot P_{row}}{P} \cdot T_{row}\left(\frac{N}{n_z \cdot P_{row}^2}\right) + T_{col}\left(\frac{N \cdot P_{row}}{P^2}\right) \end{aligned} \tag{4}$$

where $T_{row}()$ and $T_{col}()$ are functions of communication latency for the row and column communicators. The 2D processor decomposition and the symmetry requirement of the real-to-complex 3D-FFT together demands the following conditions:

$$\frac{n_z}{P_{col}} \geq 1, \quad \frac{n_y}{P_{row}} \geq 1, \quad \text{and} \quad \frac{n_x}{P_{row}} \geq 2 \tag{5}$$

P_{row} and n_z are chosen as independent variables that affect the total communication time. P_{row} can take different values depending on how the processors are arranged as a 2D processor mesh, while satisfying the validity conditions presented in Equation 5. As P_{row} decreases, P_{col} could become bigger than n_z and violates the first condition in Equation 5. However, by rotating this grid, the values of n_x and n_z can be interchanged to maintain the inequality as P_{row} decreases further. We will study this possibility in our experiments later.

4.4 Understanding the Trends in P3DFFT Performance

The total communication time in P3DFFT is impacted by three sets of variables: (i) message size, (ii) communicator size and (iii) congestion in the communicator topology. The first two variables (message size and communicator size) are directly related to the P_{row} parameter described in Equation 4. The third parameter, however, depends on the physical topology of the processes present in the

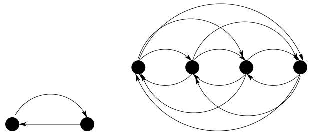


Fig. 3. `MPI_Alltoallv` Congestion Behavior

communicator and their communication pattern, as these conditions determine how many messages share the same link on the torus network.

P3DFFT internally uses `MPI_Alltoallv` to transpose the data grid. For most implementations of MPI, including the one on Blue Gene, this is implemented as a series of point-to-point communication operations, with each process sending and receiving data to/from every other process in the communicator. For this communication pattern, even in a communicator where all the processes are topologically adjacent (row communicator), the number of messages that need to traverse the same network link in the torus network can increase quadratically.

Figure 3 illustrates this behavior. Let us consider a torus of 8 × 8 × 8 processes. For a row communicator with only 2 processes, the two processes just exchange data between each other. Thus, there is only one message per link in each direction (BG/L links are bidirectional). For a row communicator with 4 processes, the exchange is more complicated with the busiest link serving up to 4 messages in each direction. Though not represented in Figure 3, it can be shown that the number of messages traversing the busiest link in each direction increases quadratically with increasing communicator size. The exception to this rule is when one dimension of the torus completes. For example, for a communicator with 8 processes, the first dimension in the torus is fully utilized; thus, since BG/L uses a 3-D torus, this would mean that these processes can use an extra wrap-around link along this dimension. In this case, the maximum number of messages on the busiest link would be half the value it would have been without this wrap-around link.

In summary, if the first dimension of the torus has a processors, for communicator sizes of 1, 2, 4, ..., a/2, a, the number of messages on the busiest link would increase as 1, 4, 16, ..., $(a/4)^2$, $(a/2)^2 \times 2$, i.e., a quadratic increase in congestion with increasing communicator size. This trend continues for the second and third dimensions as well. Using this analysis, we can observe that a small system that has a torus configuration of 8 × 8 × 8 would have a much smaller amount of congestion as compared to a large system that has a torus configuration of 8 × 32 × 16.

The top 4 graphs in Figure 5 illustrate the total bandwidth per process achieved by `MPI_Alltoallv` for different message sizes on a small system ($P = 512$). The diamonds and triangles marked on the figures show the different message sizes (and corresponding bandwidths) that are used within P3DFFT for data grid configurations of 512×512×128 and 128×512×512. We notice that as long as the message size is larger than about 1 KB, both the row and column communicator achieve the peak bandwidth; thus, for best performance, it is preferred

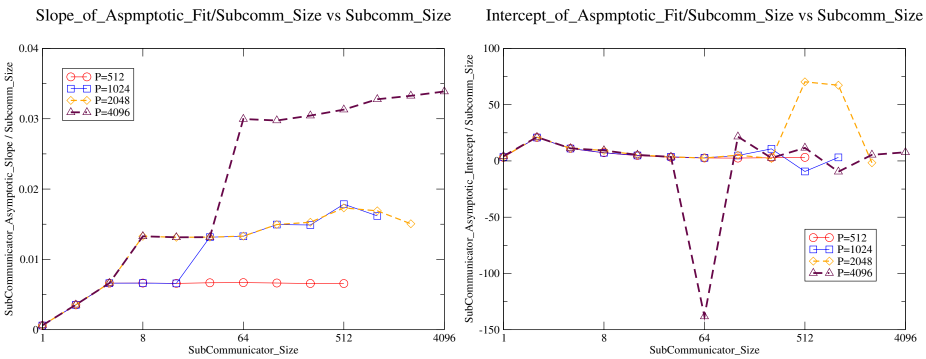


Fig. 4. Plots of scaled slope, $S()$, and intercept, $I()$, of asymptotic fit of the latency of the `MPI_Alltoallv` at P=512,1024,2048,4096. The graphs show that the slope and intercept scales with the subcommunicator size in a meaningful way.

that a large message size be used. However, as illustrated in Equation 4, when P_{row} becomes large, the message size used by the row communicator drops quadratically. This causes it to use a very small message size for large P_{row} values resulting in the network not being saturated, and consequently performance loss. Thus, a small P_{row} value is preferred. For large systems ($P = 4096$), the large impact of congestion, as described above, can be observed in the bottom 4 graphs in Figure 5. The congestion causes a two-fold difference in the `MPI_Alltoallv` bandwidths achieved by the row and column communicators.

In the network saturation region, the time taken by `MPI_Alltoallv` can be approximated as a linear function, i.e. $T_{sub}(m_{sub}) \approx \alpha_{sub} \cdot (r \cdot m_{sub}) + \beta_{sub}$ where *sub* is the sub-communicator label for either *row* or *col*, and r is the precision of the datatype that $r \cdot m_{sub}$ is the message size in byte. In order to use the asymptotic function meaningfully, we investigated how the α and β change with their corresponding sub-communicator size. The results are shown in Figure 4. Notice that the Y-axes of both pictures are divided by the size of the sub communicator. This is necessary to scale out the effect of the communicator size. Four system sizes, $P = 512, 1024, 2048, 4096$ are plotted in the figures. They all overlap nicely to some universal functions. The scaled slope and intercept functions will be called $S(P_{sub})$ and $I(P_{sub})$ respectively. They are defined as

$$S\left(P_{sub}\right) = \frac{\alpha_{sub}\left(P_{sub}\right)}{P_{sub}} \quad and \quad I\left(P_{sub}\right) = \frac{\beta_{sub}\left(P_{sub}\right)}{P_{sub}} \qquad (6)$$

For small systems $P = 512, 1024$, $S(P_{sub})$ increases linearly in $P_{sub} = 1, 2, 4$ and then becomes a constant afterward. But for system $P = 1024$ which is similar to $P = 512$, except a step jump appears from $P_{sub} = 16$ to $P_{sub} = 32$. For $P = 2048$, $S(P_{sub})$ increases linearly in $P_{sub} = 1, 2, 4, 8$, and then stays as a constant afterward. For $P = 4096$ which is similar to $P = 2048$, except with a step jump from $P_{sub} = 32$ to $P_{sub} = 64$. We believe the step jumps are due to the sudden increase of contention as P_{sub}'s topology changes as explained earlier in this section.

With Equations 6, 2 and 3, Equation 4 can be simplified to

$$\begin{aligned}T(n_z, P_{row}) &\approx \frac{n_z}{P_{col}} \left(\alpha_{row}(r \cdot m_{row}) + \beta_{row}\right) + \left(\alpha_{col}(r \cdot m_{col}) + \beta_{col}\right) \\ &= \frac{n_z}{P_{col}} \left(\frac{\alpha_{row}}{P_{row}} \frac{rN}{n_z P_{row}} + \beta_{row}\right) + \left(\frac{\alpha_{col}}{P_{col}} \frac{rN}{P} + \beta_{col}\right) \\ &= r\left[S\left(P_{row}\right) + S\left(\frac{P}{P_{row}}\right)\right]\frac{N}{P} + I\left(P_{row}\right) n_z \frac{P_{row}^2}{P} + I\left(\frac{P}{P_{row}}\right)\frac{P}{P_{row}}\end{aligned} \tag{7}$$

The $T(n_z, P_{row})$ is linear in n_z but its dependence on P_{row} is rather complicated. Since the behaviors of $S()$ and $I()$ in P_{sub} are known from Figure 4, we can reasonably describe how the total transpose time changes with P_{row}. Based on Figure 4, $S()$ is always positive and monotonic in P_{sub}. For large systems ($P = 4096$), $S() < 0.035$. For small systems ($P = 512$), $S() < 0.007$. $I()$ is more of less a positive constant of order $O(10)$ except the big negative spike occurs at $P_{sub} = 64$. For the system parameters being considered here, we are mainly interested in $P_{row} < \sqrt{P}$, each term in Equation 7 can be estimated as follows:

$$\begin{matrix}\text{1st}\\\text{term}\end{matrix} \propto \frac{N}{P} \gg P, \qquad \begin{matrix}\text{2nd}\\\text{term}\end{matrix} \propto n_z \frac{P_{row}^2}{P} < n_z < P, \qquad \begin{matrix}\text{3rd}\\\text{term}\end{matrix} \propto P_{col} < P$$

However, all the terms in Equation 7 are made equally important by $S() \ll I()$. For simplicity, let's ignore any terms that is $O(P_{row}^2)$ or higher and consider the small P_{row} limit, where $S(P_{row}) \to s_0 \cdot P_{row}$, $S(P/P_{row}) \to S_\infty$, and $I(P/P_{row}) \to I_\infty$. Equation 7 can be approximated as:

$$T(n_z, P_{row}) \approx r\left[s_0 \cdot P_{row} + S_\infty\right]\frac{N}{P} + I_\infty \frac{P}{P_{row}} \tag{8}$$

$$\Longrightarrow \quad P_{row}^{min} = P\sqrt{\frac{I_\infty}{rs_0N}} \quad \text{and} \quad T(n_z, P_{row}^{min}) \approx rS_\infty \frac{N}{P} + 2\sqrt{rs_0NI_\infty} \tag{9}$$

P_{row}^{min} is where the minimum of $T(n_z, P_{row})$ occurs.

For $N = 512 \times 512 \times 128$ and $P = 512$, $r = 4$, $s_0 \sim 0.002$, $S_\infty \sim 0.007$, $I_\infty \sim 3$, then $P_{row}^{min} \sim \sqrt{3} \sim 1.73$ and $T(n_z, P_{row}^{min}) \sim 3.5$ *msec*. For $N = 2048 \times 2048 \times 512$ and $P = 4096$, $r = 4$, $s_0 \sim 0.002$, $S_\infty \sim 0.035$, $I_\infty \sim 7$, then $P_{row}^{min} \sim 2\sqrt{7} \sim 5.3$ and $T(n_z, P_{row}^{min}) \sim 93$ *msec*. Both predicted $T(n_z, P_{row}^{min})$ values are within few percents of the actual measured experimental values.

For more accurate estimation, $O(P_{row}^2)$ terms and the full features of $S()$ and $I()$ are all needed. $I()$ has a negative spike of $O(10^2)$ at $P_{row} = 64 = \sqrt{P}$ as in Figure 4. The negative spike will certainly produce a local minimum of total transpose time for $N = 4096$. If the flat cartesian grid is rotated to increase n_z to avoid violating the validity conditions in Equation 5, P_{row} can get a lot closer to 1. Also, the discrete jumps seen in $S()$ in Figure 4 could be reflected in the observed total trasnpose time as sudden jump seen in the corresponding $S()$.

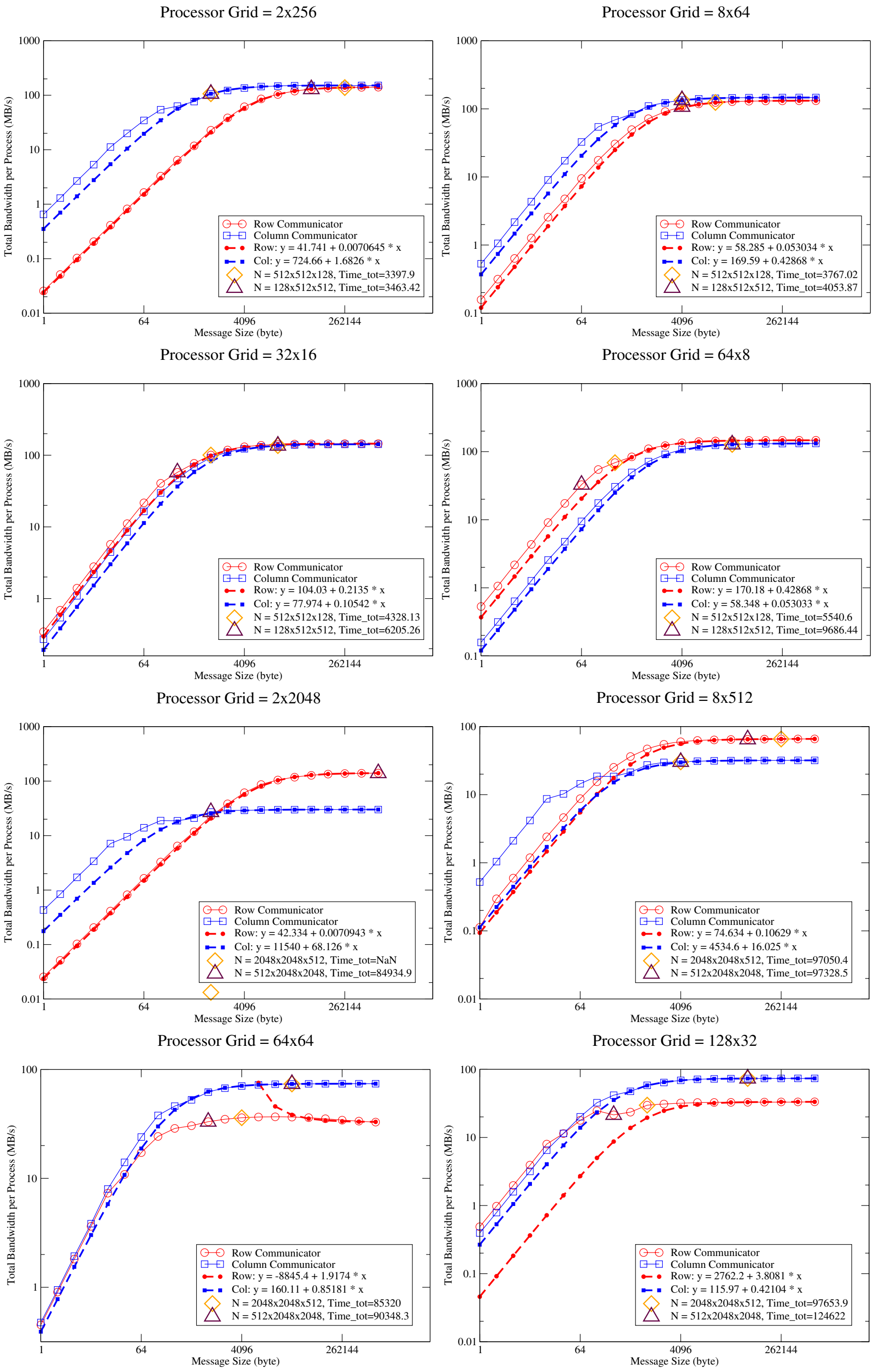


Fig. 5. Total bandwidth per process vs message size of small ($P = 512$) and large systems ($P = 4096$)

5 Experimental Evaluation and Analysis

In this section, we experimentally evaluate the P3DFFT library using a fortran physics program[1] that uses a flat cartesian mesh. Specifically, in this program, some of the variables only have x- and y-components, but no z-component. This means that the physical system emulated by this program is a quasi-2D system with preferential treatment in the z-axis. Therefore, our analysis of total communication time with respect to change in P_{row} in Equation 7 is applicable to this program, but not the communication analysis with respect to change in n_z. This is because the variation in P_{row} and P_{col} affects only the `MPI_Alltoallv` used in the two global transposes employed by the 3D-FFT which is being applied uniformly to all variables. In order words, the variation of the fortran program with respect to P_{row} is equivalent to the variation of 3D-FFT algorithm. But the variation of the fortran program with respect to n_z includes both the variation of the 3D-FFT algorithm and the special asymmetric treatment of the z-axis in the phyical problem. In Section 5.1, we first observe the communication behavior for a small half-rack (512-node) system and verify our analysis. Then, in Section 5.2, we utilize this understanding to evaluate and optimize the performance of P3DFFT for a large-scale system.

5.1 Communication Analysis on a Small-Scale System

In this section, a small BG/L system of 512 nodes is used to study the behavior of P3DFFT. These 512 nodes form a regular torus of 8×8×8 dimensions. We ran our fortran program that uses the P3DFFT library with various data grid configurations on different processor mesh arrangements. Table 1 presents the timing data from this run.

Table 1. Timing of the fortran P3DFFT program (in second) with $P = 512$ (8×8×8 torus). P: Processor mesh configuration. N: FFT data grid configuration.

P \ N	256×256×64	64×256×256	512×512×128	128×512×512
8×64	1.294	1.37	9.08	9.98
16×32	1.276	1.65	9.08	10.73
32×16	1.41	2.34	9.62	11.86
64×8	1.74		10.66	15.01

Four data grid configurations (256 × 256 × 64, 64 × 25 × 256, 512 × 512 × 128 and 128×512×512), and four different processor mesh decompositions (8×64, 16×32, 32×16 and 64×8), were attempted on the 512-node system. In Table 1, we can see that the best timing for each data grid configuration occurs at the processor-mesh with the shortest row dimension, i.e., shortest P_{row}. Also, we see that the fortran program is taking longer to finish as P_{row} increases. Both features are consistent with our findings with Equation 8 in the last section.

[1] The program, provided by Joerg Schumacher, has been modified for our benchmarking purpose.

5.2 Evaluation on a Large-Scale System

In this section, we evaluate the performance of P3DFFT on a large-scale (16-rack or 16384-node) BG/L system. Specifically, we evaluated the performance of a 2048×2048×512 data grid, with different processor-mesh configurations, on 4 racks (4096 nodes), 8 racks (8192 nodes) and 16 racks (16384 nodes). For the 4-rack system, we also tried out two different torus topologies (16×32×16 and 16×16×16). Further, we also study the impact of rotating the data grid.

Tables 7, 6, 4 and 3 show the performance results for the different system sizes and configurations with our fortran test program. All these results indicate that the best performance occurs at the smallest P_{row} and largest n_z, i.e. rotated data grid, shown in the tables, when the number of processors P and the problem size N (FFT data grid size) are fixed. The small P_{row} giving the best performance is consistent with our findings of Equation 8. The best performance occuring at largest n_z for fixed problem size N is more a feature of the physics problem being solved in the fortran program and not a feature of P3DFFT as explained in the beginning of section 5.

Tables 4 and 3 show the performance numbers of the fortran with the same problem sizes (2048×2048×512 and 512×2048×2048) and the same number of nodes (4096 nodes). The only difference between these two tables is that the different torus configurations are used. Table 4 is evaluated on a 8×32×16 torus, while Table 3 is evaluated on a 16×16×16 torus. The fastest performance at 64×64 can be explained by the big negative spike of $I()$ seen in Figure 4 and Equation 7. We notice that for the processor-mesh, 64×64, P3DFFT is 10% faster in the 16×16×16 torus as compared to the 8×32×16 torus. The reason for this behavior is the layout of the column communicator as described in Section 4.2. Specifically, in the 8×32×16 torus configuration, each row (64 processes) takes up eight physical rows on the torus. Thus, the processes in the column communicator can be up to 8 rows apart. On the other hand, in the 16×16×16 torus configuration, each row takes up only four rows, thus reducing the distance between the processes in the column communicator and consequently improving their performance.

Next, let us consider Table 6 that shows the performance for 8192 processors. If we notice the 2048×2048×512 FFT data grid configuration, we see that in this case, the smallest value of P_{row} (16×512 configuration) does not provide the best

Table 2. Timing from fortran P3DFFT program (in second) with one processor size 4096 but different torus configurations. P: Processor mesh configuration. N: FFT data grid configuration.

Table 3. P = 4096 (16×16×16 torus)

P \ N	2048×2048×512	512×2048×2048
16×256	185.9	151.2
64×64	179.2	
256×16	194.1	

Table 4. P = 4096 (8×32×16 torus)

P \ N	2048×2048×512	512×2048×2048
8×512	215.2	181.36
32×128	218.4	190.4
64×64	201.7	179.3
128×32	198.2	194.4
512×8	239.1	

Table 5. Timing from fortran P3DFFT program (in second) with two different processor sizes, 8192 and 16384. P: Processor mesh configuration. N: FFT data grid configuration.

Table 6. P = 8192 (16×32×16 torus)

P \ N	2048×2048×512	512×2048×2048
16×512	146.5	113.1
64×128	142.5	115.3
128×64	144.3	125.7
512×16	165.0	

Table 7. P = 16384 (32×32×16 torus)

P \ N	2048×2048×512	512×2048×2048
16×1024		79.6
32×512	117.9	84.4
128×128	118.1	91.1
512×32	128.2	
1024×16	160.1	

performance. Instead, 64×128 provides a better performance. This again can be explained by the big negative spike of $I()$ seen in Figure 4 and Equation 7. This suggests 8192 processor configuration is similar to the non-cubical torus 4096 processor configuration discussed earlier in Equation 8 with the existence of at least two optimal configurations. Comparing the performance impact of the data grid rotation from the $2048 \times 2048 \times 512$ configuration to the $512 \times 2048 \times 2048$ configuration, we notice that the performance improves by about 26%. Not all the improvement is from the communication time.

The final result we present is for the large 16-rack (16384-node) system. We notice that this case is a little different from the 4-rack (4096-node) and the 8-rack (8192-node) results where two optimal configurations can be obtained. However, the overall performance is still consistent with the other results. That is, performance improves with decreasing P_{row} and with increasing n_z. Specifically, reducing P_{row} can improve performance by about 15% as compared to the default 128×128 processor mesh configuration. Increasing n_z, on the other hand, can improve performance by close to 48%.

Based on all the experimental results, we notice that there could be multiple optimal system configurations, two possible ones are 1) small P_{row} in rotated data grid with larger n_z. 2) $P_{row} \simeq \sqrt{P}$ in the regular data grid with smaller n_z. The later optimal configuration may not exist in all system sizes and configurations. But the first optimal configuration seems to always exist. Rotating the FFT data grid furthers the path of performance improvements that have been stopped by Equation 5. This indicates that as we move to even larger problem sizes, the lessons learnt in this paper will have increasingly higher importance.

6 Concluding Remarks and Future Work

P3DFFT is a recently proposed implementation of parallel 3D FFT for large-scale systems such as IBM Blue Gene. While there have been a lot of studies that demonstrate the scalability of P3DFFT on regular cartesian meshes (where all dimensions are equal in length), there seems to be no previous work that studies its scalability for flat cartesian meshes (where the length of one dimension is much smaller than the rest). In this paper we studied the performance and scalability of P3DFFT for flat cartesian meshes on a 16-rack (16384-node) Blue

Gene system and demonstrated that a combination of the network topology and the communication pattern of P3DFFT can result in parts of the communication to over-saturate the network, while other parts under-utilize the network. This can cause overall loss of performance on large-scale systems. We further showed that remapping processes on nodes and rotating the FFT data grid by taking the communication properties of P3DFFT into consideration, can help alleviate this problem and improve performance by up to 48% in some cases.

While our work alleviates the issue of network saturation, it does not completely avoid it. For future work, we would like to further the study of alleviation of network contention by rotating the torus configuration through environment variable `BG_MAPPING` which allows user to rearrange process layout in the torus, and we would also like to study the impact of split-collectives to hide communication time that can be aggravated due to such saturation.

References

1. http://128.55.6.34/projects/paratec/
2. http://eslab.ucdavis.edu/software/qbox/
3. http://www.cse.scitech.ac.uk/ccg/software/dl_poly/
4. http://www.fftw.org
5. http://www.research.ibm.com/journal/rd/492/gara.pdf
6. http://www.sdsc.edu/us/resources/p3dfft.php
7. http://www.spscicomp.org/scicomp12/presentations/user/pekurovsky.pdf
8. Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation 19(90), 297–301 (1964)
9. Intel Corporation. Intel Math Kernel Library (MKL), http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm
10. Cramer, C.E., Board, J.A.: The Development and Integration of a Distributed 3D FFT for a Cluster of Workstations. In: Proceedings of the 4th Annual Linux Showcase and Conference, vol. 4. USENIX Association (2000)
11. Dubey, A., Tessera, D.: Redistribution strategies for portable parallel FFT: a case study. Concurrency and Computation: Practice and Experience 13(3), 209–220 (2001)
12. Eleftheriou, M., Fitch, B.G., Rayshubskiy, A., Ward, T.J.C., Germain, R.S.: Scalable framework for 3D FFTs on the Blue Gene/L supercomputer: implementation and early performance measurements. IBM Journal of Research and Development 49, 457–464 (2005)
13. Filippone, S.: The IBM Parallel Engineering and Scientific Subroutine Library. In: International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, London, UK, pp. 199–206. Springer, Heidelberg (1996)
14. Fitch, B.G., Rayshubskiy, A., Eleftheriou, M., Ward, T.J.C., Giampapa, M.E., Pitman, M.C., Pitera, J.W., Swope, W.C., Germain, R.S.: Blue Matter: Scaling of N-body simulations to one atom per node. IBM Journal of Research and Development 52(1/2) (2008)
15. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE 93(2), 216–231 (2005); special issue on Program Generation, Optimization and Platform Adaptation

16. Gara, A., Blumrich, M.A., Chen, D., Chiu, G.L.-T., Coteus, P., Giampapa, M.E., Haring, R.A., Heidelberger, P., Hoenicke, D., Kopcsay, G.V., Liebsch, T.A., Ohmacht, M., Steinmacher-Burow, B.D., Takken, T., Vranas, P.: Overview of the Blue Gene/L system architecture. IBM Journal of Research and Development 49, 195–212 (2005)
17. Olson, C.J., Zimnyi, G.T., Kolton, A.B., Grnbech-Jensen, N.: Static and Dynamic Coupling Transitions of Vortex Lattices in Disordered Anisotropic Superconductors. Phys. Rev. Lett. 85, 5416 (2000)
18. Schumacher, J., Putz, M.: Turbulence in Laterally Extended Systems. In: Parallel Computing: Architectures, Algorithms and Applications. Advances in Parallel Computing, vol. 15. IOS Press, Amsterdam (2008)
19. Straub, D.N.: Instability of 2D Flows to Hydrostatic 3D Perturbations. J. Atmos. Sci. 60, 79–102 (2003)
20. Klitzing, K.v., Dorda, G., Pepper, M.: New Method for High-Accuracy Determination of the Fine-Structure Constant Based on Quantized Hall Resistance. Phys. Rev. Lett. 45, 494–497 (1980)

Scalable Multi-cores with Improved Per-core Performance Using Off-the-critical Path Reconfigurable Hardware

Tameesh Suri and Aneesh Aggarwal

Department of Electrical and Computer Engineering
State University of New York at Binghamton
Binghamton, NY 13902, USA
{tameesh,aneesh}@binghamton.edu

Abstract. Scaling the number of cores in a multi-core processor constraints the resources available in each core, resulting in reduced per-core performance. Alternatively, the number of cores have to be reduced in order to improve per-core performance. In this paper, we propose a technique to improve the per-core performance in a many-core processor without reducing the number of cores. In particular, we integrate a Reconfigurable Hardware Unit (RHU) in each core. The RHU executes the frequently encountered instructions to increase the core's overall execution bandwidth, thus improving its performance. We also propose a novel integrated hardware/software methodology for efficient RHU reconfiguration. The RHU has low area overhead, and hence has minimal impact on the scalability of the multi-core. Our experiments show that the proposed architecture improves the per-core performance by an average of about 12% across a wide range of applications, while incurring a per-core area overhead of only about 5%.

1 Introduction

Recently, there has been a major shift in the microprocessor industry towards multi-core processors. Multi-core processors have considerably fewer resources available in each core. For instance, in going from a 6-way issue single core superscalar processor to a quad-core processor, the issue width in each core has to be reduced to two to keep the same die area [24]. The per-core resources are reduced to be able to integrate more number of cores on a single chip. A 56-entry issue queue on a four-way issue PA-8000 processor takes up about 20% of the die area [22]. It is obvious that a similar sized scheduler cannot be integrated in each core of a multi-core processor with large number of cores.

Fewer per-core resources degrades the performance of each thread of execution [12]. Scaling the number of cores on a chip may further reduce the per-core resources. Increasing the number of cores also exacerbates the reduction in resources by increasing die area required for peripheral hardware such as interconnects [20]. Hence, it may be argued that in a typical multi-core processor, the design choice is between increasing the number of cores with poor per-core

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 365–377, 2008.

performance and having a good per-core performance but with fewer cores on the chip.

In this paper, we propose a multi-core architecture that improves the performance of the resource-constrained cores of multi-cores with large number of cores, while maintaining the scalability of the number of cores. The per-core performance is improved by integrating an off-the-critical path reconfigurable hardware unit (RHU) in its datapath. The RHU operates entirely asynchronously with the core's datapath and is reconfigured to execute the frequently executed traces of instructions. These trace instructions do not consume the core's resources, which are then available to other instructions, effectively increasing the per-core resources, and hence the per-core performance. The proposed RHU structure also has only a small area overhead, and hence only a small impact on the scalability of the number of cores.

We also propose a novel methodology for run-time RHU reconfiguration, in which RHU reconfiguration bits are generated using a hardware/software codesign and are divided into reconfiguration instructions. The reconfiguration bits are generated at run-time to avoid recompilation of legacy code. Our experiments show that per-core performance improves by about 12% across a wide variety of applications. The area overhead to achieve the improvement is about 5% per core.

2 Multi-core Design

2.1 Basic Idea

Each core in a multi-core has an integrated RHU that is reconfigured to execute frequently executed instructions from the thread executing on the core. Once a RHU is reconfigured for a trace of instructions, those instructions are only executed on the RHU and not on the core's original datapath, thus effectively increasing the resources for other instructions. On an exception or a branch misprediction from within the trace, the execution is restarted from the start of the trace and is performed entirely on the core's original datapath. The RHU is reconfigured through dynamically generated reconfiguration bits organized into chunks of 32-bits, called *reconfiguration instructions*. We assume 32-bit long instructions in our architecture. The proposed architecture also supports static (compile-time) reconfiguration instruction generation by making these instructions visible in the external ISA. Each reconfiguration instruction consists of a specialized operation code (opcode) followed by reconfiguration bits.

2.2 Core Microarchitecture

Different instructions are simultaneously executed on the RHU and the core's original datapath. We call the original instructions (not the reconfiguration instructions) that are part of a trace executed on the RHU as RHU-instructions (*RIs*) and those executed on the core's original datapath as Proc-instructions (*PIs*). The results of RIs consumed by PIs are defined as live-outs, and those of PIs consumed by RIs are defined as live-ins. To maximize the benefit of the

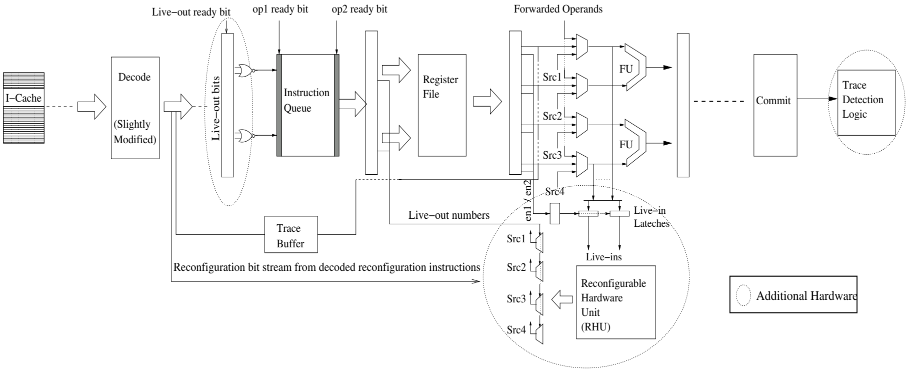


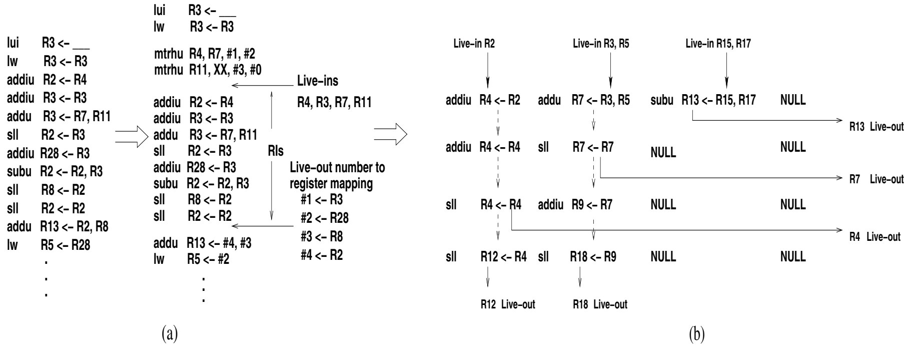
Fig. 1. Schematic diagram of the core datapath

RHU and to keep the architecture simple, RIs are selected from within the innermost loops in the applications; starting from the first instruction in the loop and forming a trace of contiguous instructions. Figure 1 shows the schematic diagram of the core datapath.

The *trace detection* logic detects traces of frequently executing instructions. Once a trace is detected, reconfiguration instructions are generated for it, as discussed later. To limit area overhead, we store the reconfiguration instructions in the thread's address space, at the end of its code section. When the start of a trace is detected in the fetch stage, its reconfiguration instructions are fetched instead of the original instructions. The reconfiguration instructions are decoded and the RHU is reconfigured using the reconfiguration bits. After the trace reconfiguration instructions are over, the fetch continues from the original instruction following the last instruction in the trace.

RHU Structure: To keep the RHU structure simple, only integer ALU operations are performed on the RHU. The most intuitive RHU structure is a two-dimensional array of interconnected ALUs. This RHU structure exploits ILP and is also able to execute strings of dependent instructions. We use a non-clocked RHU, *i.e.* the RHU is just a combinational logic. A non-clocked RHU reduces the RHU complexity, the RHU power consumption, and the overall live-in to live-out latency of the RHU. To drastically reduce the RHU-interconnect complexity, an ALU's output is forwarded only to the left operand of four ALUs in the next row. This also ensures a FO4 delay for each ALU output. The number of live-ins and live-outs per row are also limited to further reduce the RHU-complexity.

Communication between RHU and core datapath: Our approach uses a pseudo instruction – move to rhu: *mtrhu* R_s, R_t, *en1, en2* – to forward the live-in values to the RHU. This instruction is executed on the core's original datapath. When an mtrhu instruction issues, it forwards the two source registers (R_s and R_t), and en1 and en2 to the RHU, as shown in Figure 1. The R_s and R_t values are latched in the live-in latches specified by en1 and en2. Once a live-in value



```
lui    R3 <- __
lw     R3 <- R3
addiu  R2 <- R4
addiu  R3 <- R3
addu   R3 <- R7, R11
sll    R2 <- R3
addiu  R28 <- R3
subu   R2 <- R2, R3
sll    R8 <- R2
sll    R2 <- R2
addu   R13 <- R2, R8
lw     R5 <- R28
  .
  .
  .
```

```
lui    R3 <- __
lw     R3 <- R3

mtrhu  R4, R7, #1, #2
mtrhu  R11, XX, #3, #0

addiu  R2 <- R4
addiu  R3 <- R3
addu   R3 <- R7, R11
sll    R2 <- R3
addiu  R28 <- R3
subu   R2 <- R2, R3
sll    R8 <- R2
sll    R2 <- R2

addu   R13 <- #4, #3
lw     R5 <- #2
  .
  .
  .
```

Fig. 2. An example illustrating: (a) Trace of RIs along with the associated mappings, mtrhu instructions, and dependent PIs (b) Trace layout on a 4x4 RHU

is latched, it is not overwritten in the execution of that instance of the trace on the RHU.

The live-outs from the RHU are also numbered. PIs dependent on RIs are renamed to the appropriate live-out numbers. The live-out values are directly forwarded into the functional units using the live-out muxes, as shown in Figure 1. Figure 2(a) shows an example of a trace of RIs along with the live-out registers to live-out numbers mapping (this mapping is also a part of the reconfiguration instructions), the mtrhu instructions, and the renaming of the dependent PIs. This example is taken from the `applu` Spec2K benchmark. Figure 2(b) shows an example of a trace mapped on a 4x4 RHU, using a simple trace formation technique discussed later.

To wakeup PIs dependent on live-outs, each entry in the issue queue CAM portion is provided with live-out bits, equal to the maximum number of live-outs, as is shown in Figure 1. The live-out bits for live-outs required by a PI are set at dispatch, if those live-outs are not available. A ready bit is propagated in the RHU from all live-in values along all the datapaths in the RHU. Each live-out ready bit from the RHU resets the corresponding live-out bit in the CAM entries; the operand becomes ready when all live-out bits are reset.

Trace Detection: The *trace detection logic* records the most recently encountered backward branching instruction, and the outcome bits of a few following branches to detect innermost loops. When a frequently executed trace is detected from within an innermost loop, following actions are taken for the trace: (i) if it is already mapped onto the RHU or generation of its reconfiguration instructions is pending, nothing is done, (ii) if its reconfiguration instructions are already generated, they are mapped onto the RHU, and (iii) if it is an entirely new trace, its reconfiguration instructions are generated.

RHU Reconfiguration and Trace Execution: To generate longer traces, trace formation continues beyond instructions that cannot be executed on the RHU, such as complex integer and floating-point instructions, and halts when a RI candidate cannot be mapped on the RHU. The intermediate PIs are included

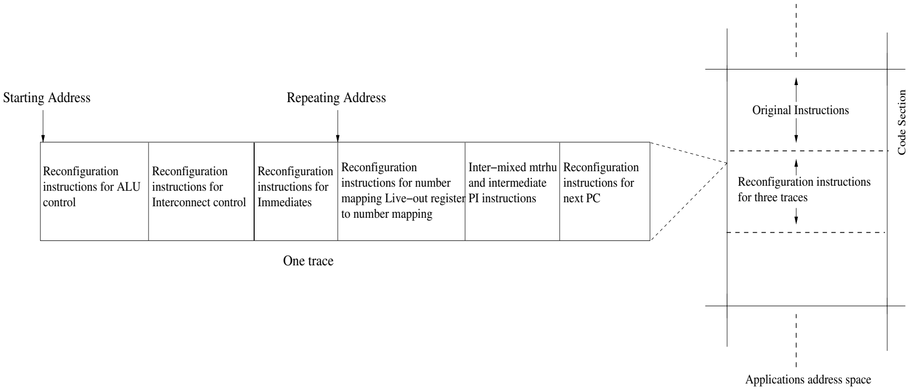

Starting Address
Repeating Address
Reconfiguration instructions for ALU control
Reconfiguration instructions for Interconnect control
Reconfiguration instructions for Immediates
Reconfiguration instructions for number mapping Live–out register to number mapping
Inter–mixed mtrhu and intermediate PI instructions
Reconfiguration instructions for next PC
One trace
Original Instructions
Reconfiguration instructions for three traces
Code Section
Applications address space

Fig. 3. Reconfiguration instruction organization

as part of the reconfiguration instructions. Load and store instructions are also executed as PIs to simplify the memory disambiguation. Figure 3 shows a trace consisting of the reconfiguration instructions and the intermediate PIs. The next PC is the address of the PI immediately following the end of the trace. The instructions for reconfiguring the RHU are executed only the first time the RHU is reconfigured for a trace. The rest of the reconfiguration instructions are required every time the trace is executed. Hence, two memory addresses – starting and repeating – are stored for each trace as shown in Figure 3.

In our implementation, we store the three most recently encountered traces; a new trace replaces the least recently used trace. We found that storing more traces did not give any noticeable improvement. The traces are stored at the end of the code section for the thread, as shown in Figure 3. An additional 3-entry *trace address* buffer is also provided to maintain the status of the traces. Every backward branching instruction's target is compared with the start PC of the traces in the trace buffer, and the execution is accordingly directed. On an exception or branch misprediction from within the trace, the execution restarts from the start of the trace, using original instructions, and is performed entirely on the core's datapath. It is important to note that the RHU cannot be reused before the current instance of the trace mapped on it commits.

2.3 Reconfiguration Instructions Generation

Performance overhead of exclusive software mechanisms to generate reconfiguration instructions is high, while area overhead of exclusive hardware mechanisms is high. In this paper, we explore an integrated hardware/software mechanism for reconfiguration instructions generation, where the reconfiguration bits are generated in hardware and then converted into reconfiguration instructions using an embedded software.

The reconfiguration instruction generation hardware includes the *trace buffer*. The trace instructions are fetched, decoded, and forwarded only to the trace buffer. The thread is not context-switched out of the core, and its instructions

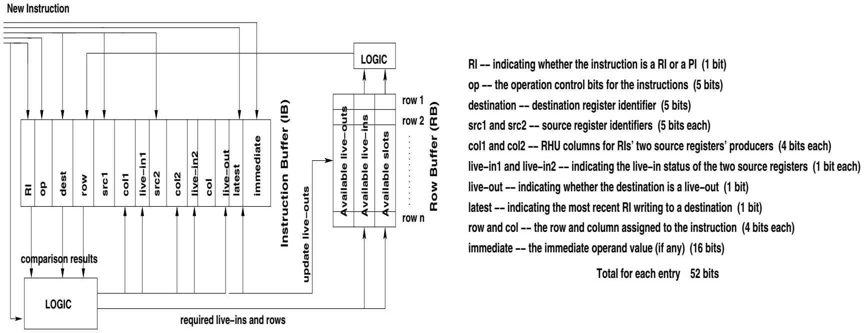

New Instruction
LOGIC
RI
op
dest
row
src1
col1
live–in1
src2
col2
live–in2
col
live–out
latest
immediate
Instruction Buffer (IB)
update live–outs
Available live–outs
Available live–ins
Available slots
row 1
row 2
row n
Row Buffer (RB)
comparison results
LOGIC
required live–ins and rows
RI -- indicating whether the instruction is a RI or a PI (1 bit)
op -- the operation control bits for the instructions (5 bits)
destination -- destination register identifier (5 bits)
src1 and src2 -- source register identifiers (5 bits each)
col1 and col2 -- RHU columns for RIs' two source registers' producers (4 bits each)
live-in1 and live-in2 -- indicating the live-in status of the two source registers (1 bit each)
live-out -- indicating whether the destination is a live-out (1 bit)
latest -- indicating the most recent RI writing to a destination (1 bit)
row and col -- the row and column assigned to the instruction (4 bits each)
immediate -- the immediate operand value (if any) (16 bits)
Total for each entry 52 bits

Fig. 4. Trace Buffer Hardware

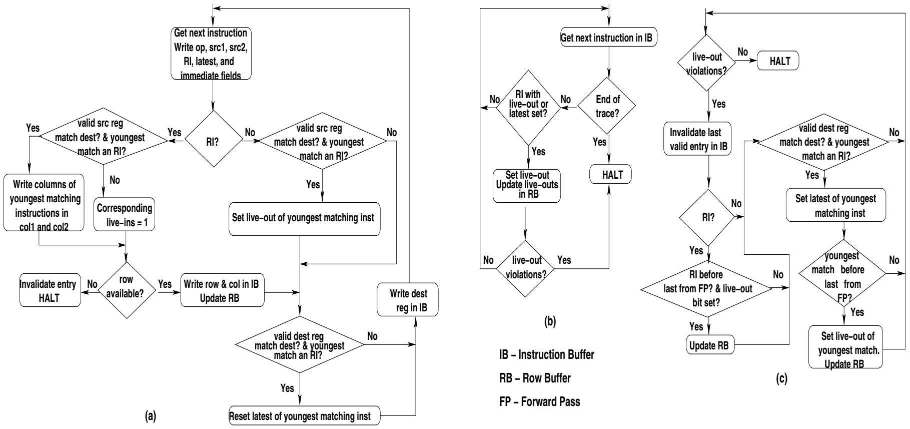

Get next instruction Write op, src1, src2, RI, latest, and immediate fields
RI?
Yes
No
valid src reg match dest? & youngest match an RI?
Yes
No
Write columns of youngest matching instructions in col1 and col2
Corresponding live–ins = 1
Set live–out of youngest matching inst
row available?
Invalidate entry HALT
Write row & col in IB Update RB
Write dest reg in IB
valid dest reg match dest? & youngest match an RI?
Reset latest of youngest matching inst
(a)
Get next instruction in IB
End of trace?
HALT
RI with live–out or latest set?
Set live–out Update live–outs in RB
live–out violations?
(b)
IB – Instruction Buffer
RB – Row Buffer
FP – Forward Pass
live–out violations?
HALT
Invalidate last valid entry in IB
RI?
RI before last from FP? & live–out bit set?
Update RB
valid dest reg match dest? & youngest match an RI?
Set latest of youngest matching inst
youngest match before last from FP?
Set live–out of youngest match. Update RB
(c)

Fig. 5. (a) Phase 1; (b) Forward pass of Phase 2; (c) Reverse pass of Phase 2; of reconfiguration bits generation

already in the pipeline continue to execute. The trace buffer hardware (shown in Figure 4) has two buffers, instruction buffer to store the instruction information and row buffer to store the availability of live-ins, live-outs, and slots in the RHU rows. We observed that a 30-entry instruction buffer, requiring about 195 bytes, is sufficient to form the traces for a RHU with 36 instruction slots. The row buffer requires about 6-12 bytes depending on the number of live-ins and live-outs per row. Additional logic is also needed to control the trace buffer operations, as discussed later in this section. To optimize the hardware for reconfiguration bits generation, only one instruction is fetched and analyzed at a time.

Reconfiguration Bits Generation: The RHU reconfiguration bits are generated in two phases as shown in Figure 5. In phase 1, the trace buffer entries are filled for each instruction. In parallel, the dependencies are resolved by comparing the operands and destinations of instructions. In this phase, the rows and columns are also allocated to instructions depending on the availability of operands. If an instruction cannot be assigned a row, its entry is invalidated and phase 1 is halted. Phase 1 for each instruction requires 3 cycles. Phase 2 starts after phase 1 and operates only on the instructions in the instruction buffer. Phase 2 has a forward pass(Figure 5(b)) and a reverse pass (Figure 5(c)) through the instruction buffer. In the forward pass, the live-outs are determined. If a live-out port is not available, then the forward pass is halted. The forward pass of phase 2 requires three cycles per instruction.

The reverse pass is used to remove any live-out violations in the forward pass. It invalidates the last valid entry in the instruction buffer, and attempts to restart the forward pass after every invalidation. If the forward pass is restarted, then reverse pass is halted and the forward pass is continued till another violation is encountered, and the process goes on. The reverse pass requires four cycles per instruction.

Reconfiguration Instructions Generation: Once phase 2 completes, embedded software (part of the operating system) instructions are fetched and executed to form the reconfiguration instructions. The embedded software reads each instruction buffer entry and generates the reconfiguration instructions. Since the embedded software is initiated by the hardware, switching to the supervisor mode may not be required. We use a specialized load instruction – *ldib* R_s *immediate* – in the embedded software to directly access the data in the instruction buffer. The immediate value specifies the instruction buffer entry to be read. The embedded software instructions are executed in-order when the all in-flight thread instructions have committed. These instructions use only the speculative register file; they do not update the architectural register files, as shown in Figure 1. These instructions do not access the memory as well. Hence, the context of the thread is intact in the core. The embedded software loads the instruction buffer entries into the core's registers. Shift and compare operations are then performed on these registers to extract the reconfiguration bits for each row. The extracted reconfiguration bits are shifted into one of the registers to form reconfiguration instructions.

3 Experimental Results

3.1 Experimental Setup

We experiment with a quad-core processor. In this paper, we experiment with non-data-sharing threads scheduled on the cores. The hardware features and default parameters of each core are given in Table 1. The per-core resources are constrained to depict a core in a multi-core processor with large number of cores, and are similar to those in the current multi-core implementations. For benchmarks, we use a collection of Spec2K and MiBench [11] benchmarks. The statistics are collected for 200M instructions after skipping 1B instructions for Spec2K benchmarks and 50M instructions for the rest. For better legibility, we present the individual results of a representative set of nine benchmarks (`art`, `equake`, `mesa`, `mgrid`, `vpr`, `sha`, `susan`, `CRC32`, and `FFT`). We evaluate the

Table 1. Experimental parameters for each core

Parameter	Value	Parameter	Value
Commit Width	4 instr.	*Instr. Window Size*	8 Int/8 Mem/16 FP
ROB Size	96 instr.	*Issue Width*	1 Int/1 Mem/2 FP
Spec. Register File	48 Int/48 FP,	*Int. FUs*	1 ALU, 1 Mul/Div, 1 AGU
Load/store buffer	40 entries	*FP FUs*	2 ALU, 1 Mul/Div
Branch Predictor	gshare 4K entries	*L2 - cache* (shared by 4-cores)	unified 2M, 8-way assoc 20 cycles
L1 - I-cache	16K direct-mapped 1 cycle	*L1 - D-cache*	16K 4-way assoc. 64 bytes block, 1 cycle

performance of each benchmark after averaging its performance with different combinations of benchmarks running on the four cores. We assume the delay in each RHU row to be equal to one CPU clock cycle.

3.2 Area Results

We integrated a 4x9 RHU with one SUN T1 OpenSource core [25] of an eight-core processor, to get the area requirement. The design was synthesized using Synopsys Design Compiler using a TSMC 90nm Standard cell library [31], and was placed and routed using Cadence SoC Encounter. After integrating the additional hardware, the core area increased by about 5%. Figure 6(a) shows the die image of the modified core.

The per core resources of the SUN T1 Opensource core may not exactly match the per core parameters given in Table 1. However, integration of the additional hardware into the SUN T1 core gives a reasonably accurate measure of the per-core area overhead of our approach in an eight-core processor. The RHU

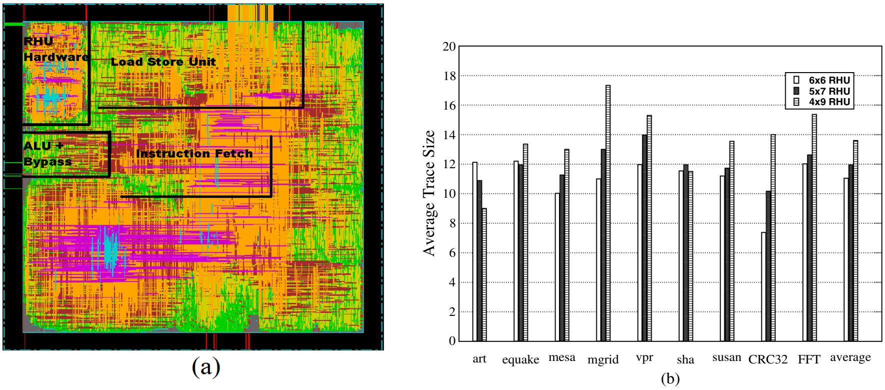


Fig. 6. (a) Die image of a Core (b) Average trace sizes of original instructions for different RHU structures

is placed close to the functional and the load/store units as the RHU interacts with them, whereas the reconfiguration bits generation hardware is placed close to the fetch/decode and the functional units.

3.3 Trace Results

We observed that two live-ins and two live-outs per intermediate row are enough to form maximum-sized traces. Row one is provided with nine live-ins and one live-out is extracted from each ALU in the last row. Our experiments showed that the average trace sizes were considerably smaller than the maximum possible 36 instructions for the 6x6 RHU. The traces were small primarily because they were terminated due to *column unavailability*, *i.e.* an instruction had to be placed in a row, but no free slots were available in that row. Hence, we also explored 5x7 and 4x9 RHUs that provide more columns than rows, while requiring almost the same number of ALUs as 6x6 RHU.

Figure 6(b) compares the trace sizes for 6x6, 5x7, and 4x9 RHUs. The 5x7 and 4x9 RHUs have significantly larger traces than the 6x6 RHU. The 4x9 RHU performs the best with an overall average trace size of about 15 instructions.

We also studied the reasons for trace termination on the best performing 4x9 RHU; the detailed results are not shown to conserve space. We observed that the traces were mostly terminated because of two reasons: *column unavailability* in the top two rows, and *row unavailability*, where a dependent needs to be placed beyond the last row. This suggests that more columns are required in the top two rows. To further increase the trace sizes, we investigate an asymmetrical RHU structure – *AsymmRHU* – for the 36 ALUs. AsymmRHU is provided 11 columns in the first and second rows, six columns in the third row, five columns in the fourth row, and three columns in a fifth row. A fifth row is added to reduce the trace terminations due to row unavailability. The live-ins and live-outs per intermediate row are kept at two. All the ALUs in rows four and five are provided with live-outs. In AsymmRHU, each ALU output is still forwarded to atmost four ALU-inputs in the next row. Figure 7(a) compares the trace sizes of AsymmRHU with the 4x9 RHU. The trace sizes increase with AsymmRHU, with the overall average reaching almost 16 instructions.

RHU Coverage: We observed that the RIs formed about 17% fraction of the overall instructions executed for the 4x9 RHU, and about 21% for AsymmRHU. The percentage of instructions executed as RIs depends on the size of the inner-most loops and the percentage of the total instructions in the application that lay within the inner-most loops.

3.4 Performance Results

Next, we present the performance (IPC) improvement of RHU-cores, with 6x6 RHU, 4x9 RHU, and AsymmRHU, over the base core. The IPC speedup is shown in Figure 7(b). Figure 7(b) also shows the IPC speedup of cores with double-sized integer scheduler. A double-sized scheduler doubles the issue queue size and

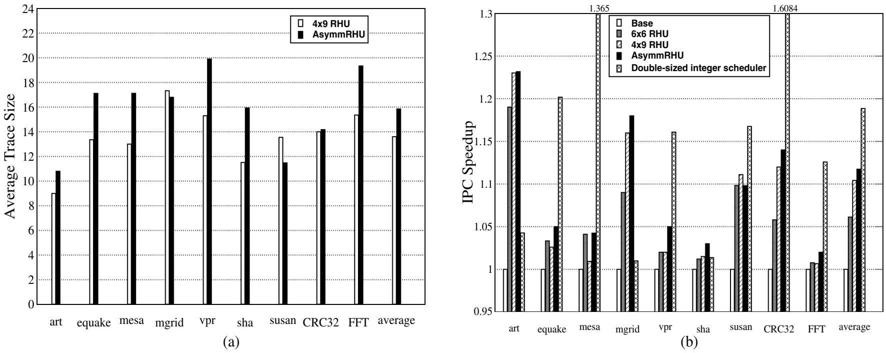


Fig. 7. (a) Average trace sizes for 4x9 RHU and AsymmRHU (b) IPC speedup of RHU-cores compared to the base core configuration and that of cores with double-sized schedulers

issue width of the base case shown in Table 1. The number of functional units are accordingly increased. Doubling the schedulers for better performance may have much higher impact on the scalability than our approach because of the increased scheduler size and additional functional units, forwarding paths and register file ports. The maximum average IPC speedup of about 12% is obtained with AsymmRHU. The 4x9 RHU achieves about 10% IPC speedup.

Interestingly, our approach performs significantly better than the double-sized scheduler configuration for `art`, `mgrid`, and `sha`. This is because the double-sized scheduler is still limited by other resources such as the fetch width, registers, etc., the pressure on which is somewhat relieved by the RHU. Additionally, when instructions are executing on the RHU, the effective issue queue size and issue width may more than double during that time. Still, the average performance of AsymmRHU is about 5% lower than the double-sized scheduler.

Our experiments showed an average of about 2 million cycles between successive reconfiguration instruction generation and an average of about 800,000 cycles between successive RHU reconfigurations. Hence, the overhead of our approach is minimal. The performance results in Figure 7(b) include the overhead of generating and executing the reconfiguration instructions.

4 Related Work

Previous studies integrate FPGA modules with a processor to improve performance. PRISM [1], Spyder [16], Piperench [5], and Garp [4] use a loosely coupled FPGA as a co-processor. Similar co-processor based proposals [18], [23], [32], [28], [30], [34] target application specific architectures.

Chimaera [13], PRISC [26], and OneChip [33] integrate the FPGA as a functional unit (RFU) in the processor datapath with direct access to the processor register file. The compiler statically generates the RFU instructions and FPGA reconfiguration bit-streams, which are used to dynamically reconfigure the FPGA. FPGAs have high area overhead, are considerably slower, and have higher energy consumption as compared to the ICs. Furthermore, FPGAs incur extensive overhead in generating and communicating the huge bit-streams required for reconfiguring them.

Other approaches execute aggregated instructions on custom functional units, for instance, [14], [15], [17], [19] fuse x86 micro-op pairs. These approaches target pairs of ALU instructions. Dynamic strands [29] extend beyond pair-wise aggregation still targeting Integer ALU instructions. The authors in [2], [3], [9] fuse a dependence chain to form a special instruction, which is then executed on non-reconfigurable custom functional unit.

Clark et al. [8] propose a restrictive reconfigurable custom compute accelerator (CCA) that has a maximum of four inputs and two outputs, executing subgraphs of a small number of instructions terminating at branch and memory instructions. The authors acknowledge the performance limitations of terminating at branch and memory instructions in [6], a restriction not present in our approach. Hence, in [6], they also propose execution of more arbitrary acyclic sub graphs that cross branch boundaries and include memory instructions. This approach requires store-load collapsing within the sub-graph, and is targeted for single-issue in-order embedded processors.

Commit time trace formation has also been proposed to improve the fetch bandwidth and perform dynamic optimizations in superscalar processors [10]. However, the reconfiguration instruction generation in our approach is significantly different from the trace formations for superscalar processors.

5 Conclusion

In a multi-core processor, scalability of the number of cores and per-core performance conflict one another. In this paper, we explore an architecture that improves per-core performance, with minimal impact on area. In the architecture, the cores have integrated reconfigurable hardware unit (RHU) to improve their performance. We propose innovative mechanisms to integrate the RHU in the core's datapath, to generate reconfiguration instructions using a hardware/software co-design, and to reconfigure the RHU. The proposed architecture improves the average per-core performance by about 12% with about 5% area overhead.

References

1. Athanas, P., et al.: Processor reconfiguration through instruction-set metamorphosis. IEEE Computer 26(3) (1995)
2. Bracy, A., et al.: Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth. In: Proc. MICRO (2004)
3. Bracy, A., et al.: Serialization-Aware Mini-Graphs: Performance with Fewer Resources. In: Proc. MICRO (2006)
4. Callahan, T., et al.: The garp architecture and c compiler. IEEE Computer 33(4), 62–69 (2000)

5. Chou, Y., et al.: Piperench implementation of the instruction path coprocessor. In: Proc. MICRO (2000)
6. Clark, N., et al.: An architecture framework for transparent instruction set customization in embedded processors. In: Proc. ISCA (2005)
7. Clark, N., et al.: Processor acceleration through automated instruction-set customization. In: Proc. MICRO (2003)
8. Clark, N., et al.: Application Specific Processing on a General Purpose Core via Transparent Instruction Set Customization. In: Proc. MICRO (2004)
9. Corliss, M.L., et al.: DISE: A Programmable Macro Engine for Customizing Applications. In: Proc. ISCA (2003)
10. Fahs, B., et al.: Performance characterization of a hardware mechanism for dynamic optimization. In: Proc. MICRO (2001)
11. Guthaus, M.R., et al.: MiBench: A free, commercially representative embedded benchmark suite. Work. Workload Characterization (2001)
12. Hammond, L., et al.: A Single-Chip Multiprocessor. IEEE Computer 30(9) (September 1997)
13. Hauck, S., et al.: The chimaera reconfigurable functional unit. In: Proc. FCCM (1997)
14. Hu, S., et al.: An Approach for Implementing Efficient Superscalar CISC Processors. In: Proc. HPCA (2006)
15. Hu, S., Smith, J.: Using Dynamic Binary Translation to Fuse Dependent Instructions. In: Int. Symp. on CGO (2004)
16. Iseli, C., Sanchez, E.: Spyder: a sure (superscalar and reconfigurable) processor. Journal of Supercomputing 9(3), 231–252 (1995)
17. Intel Corporation, Mobile Intel Pentium 4 M-Processor Datasheet (June 2003), http://www.intel.com/design/mobile/datashts/250686.htm
18. Jacob, J.A., Chow, P.: Memory interfacing an instruction specification for reconfigurable processors. In: Symp. FPGAs (1999)
19. Kim, I., Lipasti, M.: Macro-op Scheduling: Relaxing Scheduling Loop Constraints. In: Proc. MICRO (2003)
20. Kumar, R., et al.: Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In: Proc. ISCA (2005)
21. Lee, C., et al.: MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In: Proc. MICRO (1997)
22. Lotz, J., et al.: A Quad-Issue Out-of-Order RISC CPU. In: Proc. Int'l. Solid-State Circuits Conf. (1996)
23. Miyamori, T., Olukotun, K.: Remarc: Reconfigurable multimedia array coprocessor. IEICE Trans. on information and systems E82-D(2), 389–397 (1999)
24. Olukotun, K., et al.: The Case for a Single-Chip Multiprocessor. In: ASPLOS (1996)
25. Sun Microsystems, Inc. OpenSPARC T1 Micro Architecture Specification, Sun Microsystems, Inc. (2006)
26. Razdan, R., Smith, M.: A high-performance microarchitecture with hardware-programmable functional units. In: Proc. MICRO (1994)
27. Rotenberg, E.: Trace cache: a low latency approach to high bandwidth instruction fetching. In: Proc. MICRO (1996)
28. Rupp, C.R., et al.: The napa adaptive processing architecture. In: Proc. FCCM (1998)
29. Sassone, P., Wills, D.: Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication. In: Proc. MICRO (2004)

30. Singh, H., et al.: Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. IEEE Trans. on Computers 49(5), 465–481 (2000)
31. TSMC 90nm Core Library - TCBN90GHP, App. Note - Revision 1.2 (2006)
32. Vassiliadis, S., et al.: The molen polymorphic processor. IEEE Trans. on Computers 53(11) (2004)
33. Wittig, R., Chow, P.: Onechip: An fpga processor with reconfigurable logic. In: Proc. FCCM (1996)
34. Wong, S., et al.: Coarse reconfigurable multimedia unit extension. In: Proc. 9th Euromicro workshop on Parallel and Distributed Processing (1996)

TrustCode: P2P Reputation-Based Trust Management Using Network Coding

Yingwu Zhu[1] and Haiying Shen[2]

[1] Department of CSSE, Seattle University,
Seattle, WA 98122, USA
zhuy@seattleu.edu
[2] Department of CSCE, University of Arkansas,
Fayetteville, AR 72701, USA
hshen@uark.edu

Abstract. Trust management is very important for participating users to assess trustworthiness of peers and identify misbehaving peers in the open P2P environment. In this paper, we present TrustCode, a framework for P2P reputation-based trust management. Leveraging random network coding, TrustCode spreads coded feedbacks massively among peers, thereby achieving bandwidth-efficient dissemination, ensuring data availability, and yielding efficient feedback retrieval. Our simulations show that TrustCode is resilient to failures and robust against malicious nodes. To exhibit applicability of TrustCode, we also present two applications that can be built on top of TrustCode.

1 Introduction

While P2P systems have become an appealing platform to build a wide range of large-scale distributed applications (e.g., file sharing, content delivery, multimedia streaming), the open and anonymous nature of P2P systems opens the door to possible misuses and abuses of the overlay network by selfish, dishonest and malicious peers. For instance, malicious peers exploit the Gnutella overlay to spread tampered with information such as unauthenticated files and malwares (i.e., Trojan horses and viruses). Thus, reputation-based trust management, which builds trust by utilizing community-based feedbacks about past experiences of peers, has been proposed to suppress peer misbehaving, enabling peers to gauge trustworthiness of others and to selectively interact with more reputable ones.

Challenges for P2P reputation-based trust management include feedback expression, computation of trust, and storage and dissemination of trust data. Prior work [1,2,3] has well addressed the first two challenges, if not perfectly. In this paper, we focus our work on the last challenge. That is, we aim to provide efficient and robust trust data management in the P2P network where node churn is the norm and malicious peers are present. Current solutions are either centralized or distributed. Centralized solutions like eBay and Amazon, using a central server to store user feedbacks and compute user reputation, while simple, pose

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 378–389, 2008.

the problem of scalability. Prior work [1,2,3] stores (or aggregates distributed) feedbacks and computes a global trust value for each peer over P2P overlays like DHTs [4,5,6], through the notion of *trust manager* or *score manager*. For example, a trust manager for a peer i is the peer node that is responsible for the hash of i's ID. To compute the column trust vector for peers, the trust managers collaborate to aggregate distributed feedbacks and repeatedly compute the trust vector until it converges.

However, the distributed solutions have some limitations: (1) A misbehaving trust manager or peer storing feedbacks could intentionally ignores/drops some feedbacks, resulting in a "distorted" trust vector. (2) Malicious nodes could launch attacks against the trust manager or the node storing the feedbacks for the specific peer, attempting to discredit the peer or simply void the peer's reputation. (3) The number of trust managers is proportional to the number of peers in the large-scale system since the hashes of each peer's IDs are very likely uniformly distributed over the DHT. Consequently, computation of the trust vector incurs high message overhead among trust managers. Worse, if the feedbacks for a peer are stored in a different node other than its trust manager, the feedback retrieval traffic is massive. (4) Node churn could make unavailable the trust manager for a peer or the node storing the peer's feedbacks, resulting in unavailability of trust information and failure of trust computation. Although replication (e.g., by using multiple trust managers or feedback storage nodes for each peer) is able to alleviate the problem, the problem of ensuring data availability in the presence of node churn itself is still nontrivial in the P2P environment [7,8]. (5) The trust manager or the node storing the feedbacks of a popular peer (which provided a high amount of services to others) may be overburdened, incurring load imbalance.

With these research problems in mind, we propose a P2P reputation-based trust management framework called *TrustCode*, which addresses the issue of dissemination, storage, and retrieval of feedbacks in the P2P network. TrustCode has three main design goals. First, availability of trust data is emphasized because a *complete* set of feedbacks is key to computation of the trust vector that closely characterizes the peers' reputation, while node churn and presence of malicious nodes in the P2P environment are very likely to make some feedbacks unavailable. Second, retrieval of feedbacks for trust computation needs to be efficient by contacting only a small number of nodes if the feedbacks cannot be found locally on the computing node. Finally, the dissemination of feedbacks needs to be efficient and robust against node/link failures and node misbehavior; in the meantime, it needs to ensure data availability and facilitate the retrieval of feedbacks for trust computation.

To meet the goals, TrustCode exploits *random network coding* [9] to disseminate feedbacks over the underlying overlay network. In particular, we make the following contributions: (1) To the best of our knowledge, TrustCode is the first to exploit network coding in P2P trust management. By using network coding, TrustCode makes feedback dissemination bandwidth-efficient and resilient to failures. With massive distribution of coded trust information, TrustCode improves data availability and facilitates retrieval of feedbacks for trust computation. (2) TrustCode is independent of feedback expression, trust computation models, and overlay structures. It can be easily adapted to any reputation-based system as the trust information management layer. Our simulations show TrustCode has good performance in terms of dissemination speed, bandwidth cost, feedback retrieval, data availability, and failure resilience. (3) We present example applications that can be built on top of TrustCode, exhibiting applicability of TrustCode.

The remainder of the paper is structured as follows. Section 2 provides background on network coding and review of related work. We discuss TrustCode's design and its two applications in Section 3. Section 4 provides experimental setup and results. We conclude the paper in Section 5.

2 Background and Related Work

Background: Network Coding. Network coding [10] was first proposed to improve multicast session throughput. It allows an intermediate node to *create* outgoing packets by encoding its received packets instead of simply *repeating* the received packets. With random network coding [9], the encoding generates a new coded packet by the linear combination of the received packets over a Galois field $GF(2^s)$ (s with a typical value of 8 or 16), where coefficients are randomly chosen, and addition and multiplication are performed over the Galois field. Linear combination is not concatenation: the resulting encoded packet is of length of L if it is combined from a set of packets of length L. A receiver which wants to receive all the original packets, needs to receive a sufficient number of coded packets, and then performs Gaussian elimination over a matrix constructed from the coefficients and data blocks contained in each received encoded packet, to decode the original packets. The most compelling benefits of network coding are bandwidth efficiency, data dissemination simplicity, and failure resilience. Due to space constraints, please refer to [11] for more detail on network coding.

Related Work. Aberer et al. [12] proposed storing complaints as trust data in a P2P overlay P-Gid, and using replication to handle malicious nodes. In P2Prep [13], each peer aggregates others' opinions about a servant by flooding requests across the network for votes, and computes a reputation value for the servant by considering votes and credibility of the voters. Prior proposals such as EigenTrust [1], PeerTrust [2], and PowerTrust [3], are similar in the sense that they all compute a global trust vector iteratively by taking into account both feedbacks and credibility of feedback sources. Trust data are stored (and replicated) into peer nodes by the use of the underlying DHT data location mechanism. As mentioned earlier, such trust management is vulnerable to node failures and misbehavior. TrustCode differs from the prior proposals in that it exploits random network coding to disseminate coded feedbacks and stores them massively among peers. TrustCode and the prior proposals complement each other. TrustCode can provide a robust trust data management layer for the prior proposals to access the data and compute the trust vector.

3 System Design

3.1 Overview

TrustCode disseminates feedbacks across the overlay network using random network coding, which increases *diversity* of coded feedbacks among nodes and improves resilience to failures during dissemination, while achieving bandwidth efficiency. Leveraging slack storage capacity of peers, TrustCode spreads coded feedbacks massively among peers to provide data availability guarantee in the presence of node churn and malicious nodes; in the meanwhile, TrustCode is very efficient in retrieval of feedbacks for trust computation by contacting only a small number of nodes. While *informed* feedback dissemination is a plausible alternative [1], it requires coordination among nodes by exchanging information to make informed decisions. Leveraging network coding, TrustCode does not require such coordination. Each node independently makes *local* distribution decisions (i.e., simply distributing encoded packets each of which is an arbitrary combination of some packets cached locally) while still achieving fast and efficient feedback dissemination. Not defying the informed dissemination alternative, TrustCode takes a different, complementary approach to resilient and efficient dissemination over the dynamic P2P environment.

TrustCode manages feedbacks in an *epoch*-based manner. An epoch is referred to a time window of length T (e.g., days, weeks or months). The feedbacks generated within an epoch are aggregated for trust computation [2]. Consequently, dissemination, storage and retrieval of feedbacks are all epoch-based. A feedback is locally generated when two peers have done a transaction. For example, if peer A requests a service from peer B. After B provides the service, A will produce and submit a feedback for B regarding the satisfaction of the transaction. In this paper, we term peer A *feedback source* and peer B *feedback target.* A feedback contains a unique ID, epoch, timestamp, feedback source and feedback target. Due to space limitation, please refer to our technical report [14] for details. It is worth pointing out TrustCode is insensitive to feedback expression. Mentioning feedbacks here is only to ease exposition of subsequent subsections.

3.2 Packet Structure

When a peer wants to send feedback data to a neighbor node, the peer uses random network coding to create a packet by *encoding* its cached data which

[1] E.g., nodes exchange Bloom filters that contain their feedback information to make informed feedback distribution, thereby reducing bandwidth consumption over blindly random distribution.

[2] The feedbacks of multiple epochs may also be aggregated for trust computation.

target	epoch	# of feedbacks	C_1	fid_1	$\cdots$	C_k	fid_k	coded feedback block

Fig. 1. Packet format

may include original feedbacks (i.e., locally generated feedbacks or decoded feedbacks) and coded feedbacks contained in the received packets. TrustCode makes two restrictions on feedback encoding: (1) Only feedbacks within the *same* epoch can be combined; and (2) Only feedbacks for the *same* feedback target can be combined. Figure 1 shows the packet format. As an example, M is a packet for feedback target B produced by using network coding: $M = \{h(P_B), epoch, k, C_1, fid_1, \cdots, C_k, fid_k, CB\}$, where $h(P_B)$ uniquely identifies the feedback target (P_B is B's public key and $h()$ is a hash function), k is the number of feedbacks encoded in the packet, and C_i is the coefficient randomly chosen from a Galois field of a proper size for the feedback with ID fid_i. *Coded feedback block* CB is a data block encoded from the k feedbacks. Assume all original feedbacks have a same length of L. The coded feedback block also has the length of L due to network coding. Note that the TrustCode packet incurs overhead due to the list of coefficients and feedback IDs encapsulated in the packet header.

How is a packet created from received packets by using network coding? Suppose a peer caches two received packets M_1 and M_2 for feedback target B within epoch e_1: $M1 = \{h(P_B), e_1, 2, C_1, fid_1, C_2, fid_2, CB_1\}$ and $M2 = \{h(P_B), e_1, 2, C_3, fid_3, C_4, fid_4, CB_2\}$. By combining the two packets with randomly chosen coefficients a and b for M_1 and M_2 respectively, the peer produces a packet M from the coded feedbacks contained in the two packets: $M = \{h(P_B), e_1, 4, C'_1, fid_1, C'_2, fid_2, C'_3, fid_3, C'_4, fid_4, CB\}$, where $C'_1 = C_1 \otimes a$, $C'_2 = C_2 \otimes a$, $C'_3 = C_3 \otimes b$, $C'_4 = C_4 \otimes b$, and $CB = (CB_1 \otimes a) \oplus (CB_2 \otimes b)$ ($\oplus$ and $\otimes$ are addition and multiplication operations over a Galois field). CB is the coded feedback block, containing only part of information about the four original feedbacks, which means that CB alone cannot decode any of the four feedbacks. In particular, if a packet M is encoded only from a single original feedback F, then we have: $M = \{h(P_B), e, 1, C, F_c\}$, where $F_c = C \otimes F$. If $C = 1$, then $F_c = F$.

3.3 Local Storage Structure on Peers

Leveraging abundant storage capacity, each peer caches (coded) feedbacks massively to combat node failures and node misbehaving. Each peer maintains two buffers for each epoch [3]: *decoding buffer* and *decoded buffer*. Each entry in both buffers contains feedback data for a particular feedback target. When a peer A receives a packet M, it inserts M into the corresponding entry of the decoding buffer. A then may perform Gaussian elimination on this entry [4]: If there are

[3] A peer may remove the feedback data in past epochs.

[4] In our implementation, the Gaussian elimination is actually performed on a decoding matrix which is composed of the coefficients and coded feedback blocks contained in the packets in this entry.

one or more original feedbacks decoded in this process, the original feedbacks are inserted into the corresponding entry of the decoded buffer. If the cache size for each entry is limited, we may randomly pick a packet in the corresponding entry of the decoding buffer, producing a new packet from the chosen packet and received packet using network coding. Then we replace the chosen packet with the new packet in the decoding buffer. Note that we assume an infinite buffer size on nodes in this paper and leave finite buffer sizes to our future work.

3.4 Dissemination Protocol

When a new epoch starts, TrustCode spreads coded feedback data for the current epoch [5]. For feedbacks in the past epochs, TrustCode allows a peer to retrieve them from other peers. For example, a newly joined peer can not only immediately participate in coded feedback dissemination for the current epoch, but also retrieve coded feedbacks in past epochs it misses.

The dissemination protocol is fully distributed. Each peer independently makes local decisions about what to spread to its neighbors. Moreover, the protocol is an iterative network coding approach. That is, each node further divides an epoch of length T into multiple *time slices* of length t ($t \ll T$). In each time slice, each peer generates a packet encoded from the data in the decoding and decoded buffers for each neighbor and sends the packet to the neighbor. Algorithm 1 outlines the dissemination algorithm on peer x. Note that in Line 7, TrustCode sets a limit on the number of received packets and decoded feedbacks that can be encoded in one packet, in order to limit the number of coefficients and feedback IDs in the packet header and thus the packet overhead. Note that in Algorithm 1, a peer sends each neighbor only one packet for each time slice. In practice, we allow multiple packets (for different feedback targets) to be included in a single message to each neighbor peer, to speed up the dissemination process and reduce message overhead (i.e., TCP/IP header). Once an epoch ends, each peer stops dissemination of the feedbacks for this epoch [6]. However, a peer may inform its upstream neighbors of stopping sending packets if the peer deems it has already cached sufficient data; Or a peer may decides to stop dissemination if it has not received any new (or innovative) feedback data for a certain number of consecutive time slices. Algorithm 2 shows the algorithm of receiving a packet.

During dissemination, TrustCode treats locally generated feedbacks differently. If a peer generates a feedback after a transaction, the peer immediately floods the feedback (in the form of packets but with a typical coefficient of 1) to all its neighbors disregarding the time slice concept. The intuition is that TrustCode makes a few replicas of the feedback to the direct neighbors right away in case that the failures of the feedback source peer extinct the newly created feedback.

[5] We assume the Network Time Protocol(NTP) is used in the overlay network to synchronize clocks.

[6] We may allow a *grace period* for last epoch because the feedbacks generated in the very end of last epoch may not have been spreaded massively among peers.

Algorithm 1. x.disseminate()

```
$targets$ ← gather all distinct feedback targets from the decoding and decoded buffers for the current epoch
if $targets$ is empty then
    return
end if
for each neighbor node $n_i$ do
    randomly choose a target $k \in targets$
    generate a packet $p$ for the target $k$ by combining randomly chosen packets and decoded feedbacks in two buffers
    send $p$ to $n_i$ which in turn calls $receive(p)$
end for
```

Algorithm 2. x.receive(Packet p)

```
$e$ ← extract epoch # from $p$
$k$ ← extract feedback target from $p$
$CL$ ← extract the list of coefficients and feedback IDs from $p$
$CB$ ← extract coded feedback block from $p$
Insert $CL$ and $CB$ into the decoding buffer corresponding to the entry of $k$ for epoch $e$, which triggers Gaussian elimination
if any original feedback is decoded then
    insert it into the decoded buffer corresponding to the entry of $k$ for epoch $e$
end if
```

3.5 Feedback Retrieval

TrustCode allows a peer to contact other nodes to retrieve feedbacks for a specific epoch and even a specific feedback target. The contacted nodes serve the retrieval request from their decoding and decoded buffers. Upon receiving the responses, the requesting node inserts them into the decoding buffer, which triggers Gaussian elimination and thus decodes original feedbacks. As shown in our simulations, feedback retrieval in TrustCode is very efficient, contacting only a small number of nodes. Moreover, when a new peer joins the network, it can immediately populate its caches by retrieving coded feedbacks from only a small number of nodes.

3.6 Using TrustCode

Due to space constraints, we here briefly present two potential applications of TrustCode. Please refer to our technical report for details.

The first application is *trust computation using social links or votes.* With TrustCode, a peer can exploits its social relationships such as friend lists to compute the trust vector. Due to the fact that TrustCode stores coded feedbacks massively among the peers and feedback retrieval is very efficient by contacting only a small number of nodes, a small set of friend peers can recover and aggregate all the feedbacks, cooperating in trust computation by playing the role of trust managers in the prior work [1,2,3]. Leveraging TrustCode and social links, we expect the trust computation is resilient to node failures and misbehavior.

We may also use FoF (friends-of-friends) to do trust computation, splitting the load over more peers. As an alternative, the system may recruit multiple groups of peers to do trust computation. The peers in each group act as trust managers to compute a trust vector (because of TrustCode, each group should be able to recover all the feedbacks). Each group reports its trust vector as a vote. Upon receiving all the votes, we choose a trust vector which is agreed on by the majority.

The second application is *trust monitoring and versioning.* Each epoch in TrustCode represents a version and the trust data in each epoch is a snapshot of system. A trust *collector* node can gather and version the feedbacks of each epoch by contacting only a small number of nodes. A primary purpose of trust versioning is for monitoring and diagnosis. For example, if a peer consistently has a low trust value, the system may evict the peer from the network. If a large percentage of nodes in the network consistently have low trust values, the system is very likely unhealthy and needs to be alarmed. With versioning, the system keeps track of each peer's reputation history, which is important to system monitoring.

4 Evaluation

4.1 Experimental Setup

A 500-node Chord was used as the underlying overlay network. Due to large memory requirement by network coding and packet caching, we did not simulate a larger network size. The Galois field of size is 8 for random network coding. Simulations were limited to one epoch where $5,000$ feedbacks were generated for 50 feedback targets each of which on average had 100 randomly chosen nodes as feedback sources. We ran our simulations in a controlled manner: We stop dissemination of coded feedbacks, if, for *each feedback target*, there are at least $x\%$ nodes each of which has the number of cached packets (including packets in the decoding buffer and decoded feedbacks as special packets in the decoded buffer) that is equal to or larger than a threshold m ($1 \leq m \leq 100$). x and m indicate degree of distribution of coded feedbacks among peers, with a default value of 20 and 30 respectively. The size of a feedback is 312 bytes while the TCP/IP header is counted as 40 bytes for each message during dissemination.

We also simulated random feedback dissemination w/o network coding (called *Random*) for comparison against TrustCode. During dissemination, each message contains up to 5 packets (each of which is encoded from up to 10 randomly chosen packets in the node's buffers) for different feedback targets in TrustCode, while in Random each message contains up to 5 feedbacks for different, randomly chosen targets.

Three main metrics were used: (1) *# of time slices* required to reach the dissemination stop condition. It indicates how fast TrustCode spreads coded feedbacks among peers. (2) *Bandwidth cost per node.* It exposes the overhead of TrustCode in dissemination. (3) *# of nodes contacted* to recover all feedbacks. It exhibits how efficiently a peer can collect all feedbacks. It also implies level of data availability: If the # of nodes contacted is small, TrustCode provides high data availability guarantee.

After dissemination of coded feedbacks, we scheduled a *collector* node which joins the network, retrieves the feedbacks for the current epoch by contacting a set of randomly chosen nodes till it recovers all the original feedbacks, and finally leaves. $1,000$ collector nodes were scheduled in turn to perform the same operations. Then, we averaged the results in terms of *# of nodes contacted* which represents efficiency of feedback retrieval and also implies availability of trust data. If the set of randomly chosen nodes is small, then we believe feedback retrieval is efficient and trust data is highly available in TrustCode.

4.2 Results

We summarize our results before presenting the details: (1) TrustCode shows superior performance over Random which disseminates feedbacks without network coding, in terms of dissemination speed, bandwidth cost, data availability, and feedback retrieval efficiency. (2) The packet overhead in bandwidth due to the list of coefficients and feedback IDs in a packet is big (e.g., about $60 - 68.9\%$ for various values of x and m), rendering room for improvement, e.g., using compression to reduce the overhead. (3) TrustCode exhibits strong resilience to failures in dissemination and feedback retrieval. (4) TrustCode well balances coded feedback distribution among the peers (see Figure 3(c)). Due to space constraints, some data are omitted and please refer to our technical report for more details of the results.

Performance with Different Configurations. The first set of experiments investigate performance of TrustCode under different dissemination stop conditions. Figure 2 shows performance of TrustCode for various values of x and m. We can see that more massive distribution of coded feedbacks (bigger x and m) makes feedback retrieval more efficient and data more available, but at the expense of more time slices and higher bandwidth consumption. Contacting a small number of randomly chosen nodes is able to recover or decode all original feedbacks. This is very encouraging, especially for the P2P settings where node

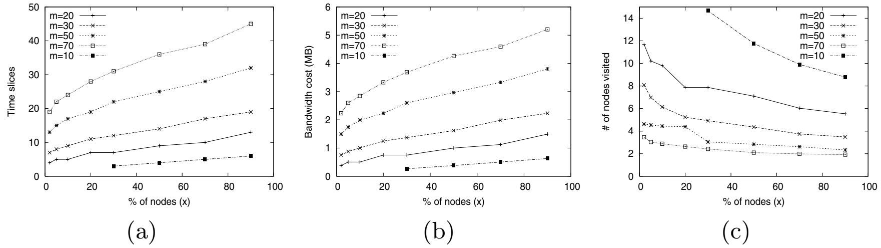
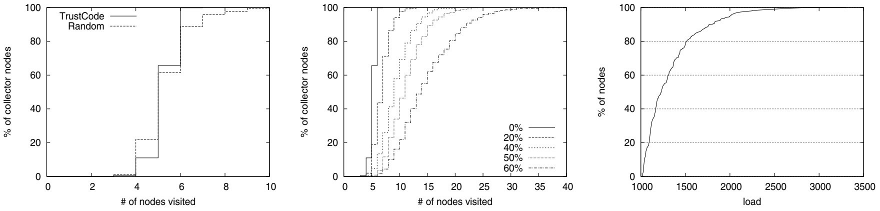


Fig. 2. Performance of TrustCode for various values of x and m. (a) # of time slices required to reach the dissemination stop condition. (b) Bandwidth cost per node. (c) # of nodes contacted to decode all the original feedbacks.



(a) (b) (c)

Fig. 3. $m = 30$, $x = 20$. (a) CDF of collector nodes. (b) CDF of collector nodes for various percentages of malicious nodes. (c) CDF of nodes with respect to load (defined as the number of packets in two buffers). The mean load of peers is $1,331$ and standard deviation is 318.

churn and failures are the norm. When $m = 30$ and $x = 20$ (see Section 4.1 for specifications), a small number 5-6 of arbitrarily chosen nodes are able to recover all the original feedbacks, showing strong resilience to failures and high efficiency in feedback retrieval, while incurring low bandwidth cost in dissemination. The main reason for such high data availability and feedback retrieval efficiency is due to network coding which maximizes diversity of coded feedbacks among peers. Figure 3(a) shows CDF of $1,000$ collector nodes with respect to # of nodes contacted in order to retrieve all the original feedbacks. We can see that at most 7 nodes are needed in order to decode all original feedbacks. In the rest of the paper, we focus on TrustCode with $m = 30$ and $x = 20$ unless specified otherwise.

Comparison with Random. We compared Random against TrustCode ($m = 30$ and $x = 20$) when they have similar level of data availability — that is, a same small number of nodes are able to recover all original feedbacks, thus providing same degree of data availability and retrieval efficiency. Table 1 presents the comparison when *# of nodes contacted* for TrustCode and Random is 5.23 and 5.33 respectively with $1,000$ collector nodes tested. Note that Random needs to replicate feedbacks of each feedback target with $m = 86$ and $x = 55$ in order to yield the similar degree of data availability and retrieval efficiency. This means that each peer in Random needs to devote more storage to cache feedbacks and more bandwidth to disseminate feedbacks. As shown in Table 1, TrustCode achieves one order of magnitude faster dissemination of feedbacks than Random and saves about 73% bandwidth over Random. Figure 3(a) plots CDF of collector nodes for TrustCode and Random. TrustCode has superior performance over Random because it spreads coded feedbacks more diversely and thus are more resilient to failures.

Impact of Malicious Nodes. In this set of experiments, we explored the impact of malicious nodes on TrustCode. Malicious nodes receive coded feedbacks but do not disseminate any coded feedbacks during dissemination, and they do not respond to feedback retrieval requests (from collector nodes). The parameter

Table 1. Performance comparison

	# of nodes contacted	*# of time slices*	*Bandwidth (MB)*	m	x
TrustCode	5.23	11	1.25	30	20
Random	5.33	138	4.59	86	55

Table 2. Impact of malicious nodes ($m = 30$, $x = 20$)

	% of malicious nodes						
Metrics	0	10	20	30	40	50	60
# of time slices	11	12	13	15	17	21	25
bandwidth cost (MB)	1.25	1.35	1.44	1.65	1.85	2.26	2.63
# of nodes contacted	5.23	5.9	6.87	8.13	9.43	11.24	14.79

x in the dissemination stop condition represents redundancy of coded feedbacks among peers: In the presence of malicious peers, we disregard coded feedbacks in malicious nodes, and x represents redundancy of coded feedbacks for each feedback target among only *benign* peers. Table 2 shows performance of TrustCode with respect to various fractions of malicious nodes. Figure 3(b) plots CDF of $1,000$ collector nodes with respect to different percentages of malicious nodes. TrustCode shows strong resilience to failures in dissemination with modest increase in bandwidth cost, and it is very efficient in feedback retrieval even when a large percentage of nodes are malicious. The driving reason for failure resilience is the diversity of coded feedbacks among peers by using network coding.

5 Conclusions

Exploiting network coding and storage capacity on peers, TrustCode provides a framework to manage trust data in P2P networks. Thanks to diversity of coded feedbacks cached among peers, TrustCode exhibits strong resilience to failures and high efficiency in feedback retrieval. Our simulations show that TrustCode is able to distribute coded feedbacks across the network in a fast, failure-resilient, and bandwidth-efficient manner. By contacting a small number of random nodes, TrustCode is capable of recovering all the feedbacks, ensuring high data availability.

References

1. Kamvar, S.D., Schlosser, M.T., Garcia-Molina, H.: The eigentrust algorithm for reputation management in P2P networks. In: Proceedings of the 12th international conference on World Wide Web (WWW), New York, NY, USA, pp. 640–651 (2003)
2. Xiong, L., Liu, L.: PeerTrust: Supporting reputation-based trust for peer-to-peer electronic communities. IEEE Transactions on Knowledge and Data Engineering 16(7), 843–857 (2004)

3. Zhou, R., Hwang, K.: PowerTrust: A robust and scalable reputation system for trusted peer-to-peer computing. IEEE Trans. Parallel Distrib. Syst. 18(4), 460–473 (2007)
4. Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of ACM SIGCOMM, San Diego, CA, pp. 149–160 (August 2001)
5. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Proceedings of the 18th IFIP/ACM International Conference on Distributed System Platforms (Middleware), Heidelberg, Germany, pp. 329–350 (November 2001)
6. Zhao, B.Y., Kubiatowicz, J.D., Joseph, A.D.: Tapestry: An infrastructure for fault-tolerance wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley (April 2001)
7. Chun, B.-G., Dabek, F., Haeberlen, A., Sit, E., Weatherspoon, H., Kaashoek, M.F., Kubiatowicz, J., Morris, R.: Efficient replica maintenance for distributed storage systems. In: Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI) (May 2006)
8. Haeberlen, A., Mislove, A., Druschel, P.: Glacier: Highly durable, decentralized storage despite massive correlated failures. In: Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, Massachusetts (May 2005)
9. Ho, T., edard, M., Shi, J., Effros, M., Karger, D.: On randomized network coding. In: Proceedings of 41st Annual Allerton Conference on Communication, Control and Computing (October 2003)
10. Ahlswede, R., Cai, N., Li, S.-Y.R., Yeung, R.W.: Network information flow. IEEE Transactions On Information Theory 46 (July 2000)
11. Fragouli, C., Boudec, J.-Y.L., Widmer, J.: Network coding: an instant primer. SIGCOMM Comput. Commun. Rev. 36(1), 63–68 (2006)
12. Aberer, K., Despotovic, Z.: Managing trust in a peer-2-peer information system. In: Proceedings of the tenth international conference on Information and knowledge management, pp. 310–317 (2001)
13. Cornelli, F., Damiani, E., di Vimercati, S.D.C., Paraboschi, S., Samarati, P.: Choosing reputable servents in a P2P network. In: Proceedings of the 11th international conference on World Wide Web, New York, NY, USA, pp. 376–386 (2002)
14. Zhu, Y.: TrustCode: P2P reputation-based trust management using network coding, tech. rep., Department of CSSE, Seattle University (June 2007)

Design, Analysis, and Performance Evaluation of an Efficient Resource Unaware Scheduling Strategy for Processing Divisible Loads on Distributed Linear Daisy Chain Networks

Bharadwaj Veeravalli[1] and Jingxi Jia[2]

[1] Senior Member, IEEE, IEEE-CS
[2] CNDS Lab, Department of Electrical and Computer Engineering,
The National University of Singapore, Singapore 117576
{elebv,g0500093}@nus.edu.sg

Abstract. In this paper, we present a generalized and novel load distribution strategy for scheduling divisible loads on linear networks, when speeds of the computing nodes and the communication links are unknown a priori. This strategy, which is referred to as *Wait and Compute Strategy* (WCS), uses a portion of the total load to estimate speed parameters and then use processors in an iterative manner in phases to minimize the overall processing time. We present an analysis of this strategy with respect to time performance and compare its performance with the previous works both analytically and through simulation studies under several influencing parameters. From our findings, we will show that the proposed generalized strategy achieves a better performance in most cases.

Keywords: Divisible Loads, Linear Networks, communication delays, processing time.

1 Introduction

Handling computationally intensive tasks on a networked set-up is a challenging problem. The complexity of the problem becomes multi-fold when computing and communication resource capabilities are unknown during the scheduling phases. Divisible load paradigm has been proposed as an effective technique in handling large scale computationally intensive tasks on networks. Initial studies were done by Cheng and Robertazzi [1] in 1988, and later the theory was formally referred to as *Divisible Load Theory* (DLT) [2]. DLT proposes elegant solutions, optimal in many cases with regular graphs, to handle large scale processing loads on networks using a linear (and also affine) model. The computation and communication delays in the nodes and links are explicitly captured in the problem formulation to seek optimal solutions. A recent survey article [3] present important results from the literature.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 390–401, 2008.

In the recent past the works in DLT literature emphasized on application oriented research as well as fundamental research on a fair basis. In the application oriented research, several application problems are shown to have direct use of DLT paradigm in seeking efficient fast solutions. Some related works that can be mentioned are multimedia applications [4], large-scale database search problems [5], and use of DLT paradigm with clusters of workstations [6]. On the other hand, work pertaining to fundamental research includes design of efficient strategies with buffer constraints [7], considering communication start-up cost [8], and scheduling divisible loads with multiround algorithms [9].

Most works referred to above are based on a fundamental assumption that the computing and communication capabilities of the resources (nodes and links) are known a priori to the scheduler which facilitates to generate an optimal, if not, a feasible schedule. Only one of the earlier works [10] digressed from this assumption and considered a time-varying nature of processor speeds and link speeds for a bus topology. A recent work [11] proposes a strategy in which speed parameters are estimated as opposed to the idea/assumption in [10] wherein the time-varying nature of the speeds is also assumed to be known in advance. However, the work in [11] is primarily designed for bus-like network, and it seems to be inapplicable in the context of linear networks. Work reported in [12] extends the previous work to linear networks using a greedy methodology. Below, we shall highlight our key contributions in this paper.

In this paper, we propose a generalized scheduling strategy for processing divisible loads on linear networks. The challenge lies in tackling the scheduling problem when the processor and link speeds are unaware to the scheduler. Our proposed strategy, which is referred to as *Wait and Compute Strategy* (WCS), works in an incremental fashion, in phases, to progressively accommodate processors as and when they respond to probe messages. At present, in the literature, there are only two works that addresses this problem of scheduling divisible loads under unknown network conditions (speeds of processors and links). The strategy proposed here attempts to generalize the methodologies proposed so far in the literature. While algorithms PSD and PCD in [11] are very much akin to bus networks, algorithm ESS proposed in [12] is a greedy approach. We present a rigorous analysis of WCS and quantify its performance with respect to several influencing parameters.

The paper is organized as follows. In Section 2, we introduce the mathematical model and useful notations that are used throughout this paper. We also introduce our motivation in this section. In Section 3 we present the design and the analytical results of WCS. Simulation results and relative study are included in Section 4, and Section 5 concludes the paper.

2 Mathematical Model, Notations and Motivations

A linear network with processing nodes and communication links is shown in Figure 1. Each node or processor is equipped with a front-end processor which off-loads the communication responsibilities of that processor. This enables

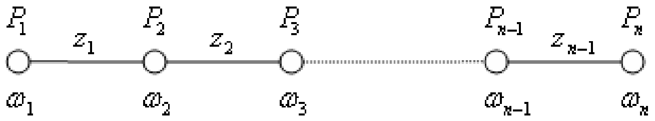


Fig. 1. Linear Daisy Chain Network Architecture with n processors and $(n-1)$ links

computation and communication to be carried out simultaneously [13]. We also assume that each node has adequate buffers to hold and process the data.

The total load to be processed is initially stored on the root processor P_1. In this setting, we assume that the computing speeds of the nodes (except the root processor, where our scheduler resides) and communication delays of the links are not known in advance. Further, a linear cost model for communication and computation is adopted as in the literature. Thus our objective is to minimize the total processing time of the entire load (time to complete processing from $t = 0$) under the above assumptions.

Now, we define some notations that will be used throughout this paper. We define L to be the total load to be processed and L_i to be the amount of load to be processed in the i^{th} computation phase. We will explain and define the computation phase in the next subsection. We use α_j^i to denote the fraction of the load L_i dispatched to the j^{th} processor P_j during the i^{th} computation phase. Notations ω_i, z_i, T_{cm} and T_{cp}, are widely defined in the literature [2]: $\omega_i(z_i)$ is the ratio of the standard processor(link) speed to the processor P_i(link l_i) speed; $T_{cp}(T_{cm})$ is the time taken by a standard processor(link) to process(communicate) one unit of load. Finally, we define η as the fraction of the total load used in the probing phase as a Probing Load (PL). A PL is a load fraction which is used to probe the processor and link speeds.

2.1 Motivation

To schedule divisible loads in resource unaware environments, the work [11] proposed several algorithms with probing technique. However, the algorithms in [11] have a continuous dispatching nature, which has an adverse effect in linear network set-up, as processors may get overloaded because of continuous dispatching. To solve the above problem, [12] proposed a phase based algorithm *Early Start Strategy* (ESS). ESS uses a two-phase approach - a probing phase (PP) followed by several computation phases. At the beginning of the PP, the root processor P_1 sends a probing load (PL) to P_2. P_2 will start to compute PL immediately after it receives PL, while at the same time it sends a copy of PL to P_3. This process continues with every processor. Each processor will record two important time stamps – the time it finishes receiving PL (denoted as T_i^c) and the time it finishes processing PL (denoted as T_i^p), and will send back these time stamps through a processing task completion (PTC) message to P_1

when it finishes processing PL. From this information, P_1 can estimate z_i and ω_i. Furthermore, as PTC is short message, the transmission time is negligible.

Notice that the arrival of PTC messages is arbitrary in time. To make full use of the first processor to respond, in ESS, the scheduler divides the remaining part of load (total load minus PL) into several parts $(L_1, L_2, ..., L_m)$, and the processing time of each part is referred to as one computation phase, denoted as $phase_1, phase_2, ..., phase_m$. After the first PTCs has been received, P_1 will apply divisible load scheduling paradigm on the first part of the load [1] for this processor and itself. During $phase_1$, it is possible that other processors may respond to their processing of PL via PTC messages one after another. Now, to accommodate these processors, P_1 employ divisible load paradigm again for all the detected processors at the end of $phase_1$, and that triggers $phase_2$. This recursive way of working continues until the entire load has been taken up, or all the processors have been detected by P_1. In the latter case, P_1 will dispatch the remaining load to all the processors.

As described above, we can see that ESS is a greedy strategy which takes distinct advantage in utilizing the first processor to respond. Other fast processors, which return their PTCs during $phase_1$, should wait till the beginning of $phase_2$. It will not be a problem for those processors responding a bit earlier than the $phase_2$'s start time. However, those processors responding a little bit later than the first responded processor will suffer a nearly entire phase idle time. This is a waste of computation power. A natural idea is waiting for more responses before starting $phase_1$, hence derives the name *Wait-and-Compute Strategy* (WCS).

3 Design of Wait-and-Compute Strategy

In WCS, we introduce a parameter k, which controls how many responses WCS will wait before triggering the $phase_1$. In a very special case, when $k = 1$, WCS becomes ESS. However, when k is larger than 1, WCS will attempt to wait for more processors returning their PTCs, before starting $phase_1$. This is in the hope of accumulating more processing power to accommodate the load in the initial phase. An interesting and noteworthy point at this juncture is as follows, WCS (When $k > 1$) adds some idle time before starting the computation. This later start approach by WCS may be compensated with the presence of additional fast processors whose PTC would have been just-in-time when the $phase_1$ of ESS would have started. This has both advantages and disadvantages as shown via our simulation studies later.

Assuming that WCS waits for k processors, and P_l is the last processor to issue its PTC in this k processor set. Using set A_{phase1} to denote these processors, we have $A_{phase1} = \{P_1\} \cup \{P_j \mid T_j^p < T_l^p,\ j = 2, \ldots, n\}$ and $|A_{phase1}| = k + 1$. Notice that, P_1 cannot obtain the respective link speeds. However, if we align all processors in A_{phase1} in an ascending order according to their subscripts, P_1 can

[1] Since in ESS, P_1 will start to compute the rest of load immediately after it sends out PL, the first part of load is actually equal to L_1 minus the load consumed by P_1 before the first PTC comes.

estimate the cumulative communication delay between P_i ($P_i \in A_{phase1}$) and its adjacent processor P_j ($i < j$) in A_{phase1} [2], denoted as Z_j, by (1).

$$Z_j = \sum_{i}^{j-1} z_k = \frac{T_j^c - T_i^c}{\eta L \cdot T_{cm}} \tag{1}$$

To clarify this point, let $k = 2$, and P_i and P_j ($i < j$) be the first two responded processors in the linear network except the root. After P_1 receives the PTCs from P_i and P_j, it will trigger $phase_1$. The remaining load of L_1, which is equal to $(L_1 - \frac{\max(T_i^p, T_j^p)}{KT_{cp}})$ [3], will be distributed among P_1, P_i and P_j following the optimality principle [2]. This principle states that the load should be distributed in a manner such that all participating processors (within the current phase) should stop computing at the same time instant. To obtain the load distribution, P_1 should solve the following recursive equations,

$$\alpha_1^1 KT_{cp} = (\alpha_i^1 + \alpha_j^1) Z_i T_{cm} + \alpha_i^1 \omega_i T_{cp} \tag{2}$$

$$\alpha_i^1 \omega_i T_{cp} = \alpha_j^1 Z_j T_{cm} + \alpha_j^1 \omega_j T_{cp} \tag{3}$$

together with

$$\alpha_1^1 + \alpha_i^1 + \alpha_j^1 = 1 \tag{4}$$

Solve (2) - (4) to obtain α_1^1 as,

$$\alpha_1^1 = \frac{\lambda(1+\gamma)Z_i + \gamma\omega_i}{\lambda(1+\gamma)Z_i + \gamma\omega_i + K(\gamma+1)} \tag{5}$$

where,

$$\gamma = \frac{\omega_j + \lambda Z_j}{\omega_i}, \qquad \lambda = \frac{T_{cm}}{T_{cp}} \tag{6}$$

Then, P_1 can estimate the time consumed for processing in $phase_1$ is given by,

$$T_1 = K(L_1 - \frac{\max(T_i^p, T_j^p)}{KT_{cp}}) \frac{\lambda(1+\gamma)Z_i + \gamma\omega_i}{\lambda(1+\gamma)Z_i + \gamma\omega_i + K(\gamma+1)} T_{cp} \tag{7}$$

Therefore, P_1 knows when to trigger $phase_2$. Notice that before the end of $phase_1$, more processors may have returned their PTCs. They will be engaged in computation in $phase_2$ together with the processors in $phase_1$. Thus, at the beginning of $phase_2$, P_1 will solve the recursive equations for all available processors to obtain the optimal load distribution (within the current phase).

The above process continues until either all the processors have been detected and used or the last load has been dispatched. Thus, supposing there are totally m phases, the overall processing time of the entire load is

$$T_{overall} = T_i^p + T_1 + T_2 + \ldots + T_m \tag{8}$$

where T_j is the duration of $phase_j$, and T_i^p is the k^{th} PTC response time.

[2] P_j and P_i may not be the adjacent processors in the underlying network.

[3] Part of L_1 is processed. This is because P_1 is allowed to start computing at time $t = 0$, the same as ESS [12].

From what we have discussed above, we notice that the key difference between ESS and WCS (When $k > 1$) exists in the $phase_1$. ESS triggers an earlier start of $phase_1$, while WCS has more computation power in $phase_1$. Thus, it is interesting to compare the $phase_1$'s finish time of ESS and WCS.

Let the first two processors returning their PTCs be P_j and P_i such that $T_j^p < T_i^p$. From [12], we can obtain the time when ESS finishes $phase_1$, denoted as $T_{phase_1}^{ESS}$, by (9)

$$T_{phase_1}^{ESS} = \theta_{ESS}(L_1 - T_j^p / KT_{cp}) + T_j^p \tag{9}$$

where θ_{ESS} is,

$$\theta_{ESS} = KT_{cp} \frac{\lambda z_{1j} + \omega_j}{\lambda z_{1j} + \omega_j + K} \tag{10}$$

Notice that z_{1j} is defined as the cumulative communication delay between P_1 and P_j.

On the other hand, when $k = 2$, from the discussion above, we can obtain the time when WCS finishes $phase_1$, denoted as $T_{phase_1}^{WCS}$, by (11),

$$T_{phase_1}^{WCS} = \theta_{WCS}(L_1 - T_i^p / KT_{cp}) + T_i^p \tag{11}$$

where θ_{WCS} is,

$$\theta_{WCS} = KT_{cp} \frac{\lambda(1+\gamma)Z_i + \gamma\omega_i}{\lambda(1+\gamma)Z_i + \gamma\omega_i + K(\gamma+1)} \tag{12}$$

Then, by equating the expressions (9) and (11), we can obtain the critical size for L_1, denoted as L_1^c, by (13).

$$L_1^c = (\frac{\theta_{ESS}T_j^p - \theta_{WCS}T_i^p}{KT_{cp}} + T_i^p - T_j^p)/(\theta_{ESS} - \theta_{WCS}) \tag{13}$$

When $L_1 = L_1^c$, WCS and ESS will have exactly the same performance. Further, since WCS has more computation power in $phase_1$, when $L_1 > L_1^c$, WCS will finish $phase_1$ earlier. On the other hand, when $L_1 < L_1^c$, WCS does not have enough time to catch up with ESS, and hence ESS will finish $phase_1$ earlier.

Remarks: A strategy that finishes $phase_1$ earlier does not guarantee to finish the whole load earlier. However, processors which are engaged or respond in $phase_1$ are fast processors. Finishing $phase_1$ earlier implies starting $phase_2$ earlier with most fast processors in the network, and hence will highly probably have a shorter overall processing time. This is verified by our simulation later. Further we assume $k = 2$ for WCS in above analysis, but this does not mean $k = 2$ is the best choice. In the next section, we conduct experiments to identify the best possible value of k with respect to certain information about the network.

4 Simulation Results and Discussions

Since the algorithms in [11] are not applicable in linear networks, in this section, we present simulation tests to compare the performance of ESS, WCS and a conservative DLT strategy (denoted as "pureDLT"). In the pureDLT strategy, P_1 will wait until receiving all the PTCs, and then start to compute. However, before that, we further discuss some key assumptions on the choice of certain parameters and networks below.

The first issue is on our decision of parameter η. one may prefer a small η, as a larger η indicates more unimportant computation. However, a too small probing load cannot accurately detect the speed parameters, as there may be small-term perturbations of processor and link speeds. In this sense, large η is needed. Thus, to strike a balance, we let parameter η fall into the range $(0.03, 0.07)$, which is seen to be appropriate in our experiments.

The second issue is on partitioning the total load prior to the load distribution. Since more and more processors will be engaged in the computation work as time progresses, to fully exploit this property, the total load can also be divided in an increasing fashion. One such load partition that satisfies such a property is as follows. For an n processor system, the total load is partitioned as follows and a partition L_k is to be distributed in $phase_k$ among processors in the set A_k, respectively.

$$L_i = (2^i + 1)(L - \eta L - L_1)/\sum_2^m (2^k + 1) \quad i = 2, \dots m$$
$$L_1 = \varepsilon * L$$

where $m = \lceil log_2 n \rceil + 1$. Notice that ε is a parameter of the load distribution, which defines the size of L_1. We refer to the above load partitions as Π[4].

Let $L = 100$, z and ω fall into the ranges $(0.2, 0.7)$ and $(2, 7)$ respectively, and the root processor has an average speed given by, $\omega_1 = K = 4.5$. We let $T_{cm} = 1$ and $T_{cp} = 2$. For WCS, we first assume $k = 2$. We refer to the processing time for ESS, WCS, and pureDLT as T-ESS, T-WCS, and T-pureDLT, respectively. Notice that the "pureDLT" serves as an upper bound on the time performance of our strategy. We denote the fraction of the fast processors and links in the network by r_f[5]. All the parameters in our simulation experiments are generated in a random fashion following a uniform distribution in their respective ranges. Each category of experiments is repeated 25 times and an average values are reported. Tables 1 and 2 show the results of our experiments. We will now present our results that demonstrate the influence of parameters η, ε and n.

Effect of η : Parameter η fundamentally determines the size of PL. Although all strategies' processing times increase as η increases, the increase in the processing

[4] Notice that as we allow P_1 to participate in computation, when it is very fast, it may even finish computing L_1 before the first PTC. Then we let it to compute L_2 and re-index L_2 to L'_1,$\dots$ L_k to L'_{k-1}, and so on.

[5] We designate a processor and a link as fast when their speeds fall in $(0.2, 0.3)$ and $(2, 3)$, respectively, as smaller numbers represent faster links/processors and vice-versa.

Table 1. Experimental Results when $r_f = 0.75$

ε	n	$\eta = 0.03$			$\eta = 0.07$		
		T-ESS	T-WCS	T-pure	T-ESS	T-WCS	T-pure
0.07	5	189.70	185.50	195.70	203.43	202.99	228.34
	8	173.73	169.29	182.32	190.06	188.51	221.19
	13	161.83	156.27	177.48	175.17	173.43	227.45
	20	157.01	153.60	182.86	170.99	171.31	242.57
0.12	5	194.77	186.59	195.70	207.20	203.49	228.34
	8	182.41	172.32	182.32	192.60	189.56	221.19
	13	171.99	159.49	177.48	182.02	174.18	227.45
	20	167.27	156.22	182.86	176.98	173.06	242.57

Table 2. Experimental Results when $r_f = 0.25$

ε	n	$\eta = 0.03$			$\eta = 0.07$		
		T-ESS	T-WCS	T-pure	T-ESS	T-WCS	T-pure
0.07	5	244.65	241.87	249.71	278.81	282.13	283.45
	8	220.59	219.12	228.85	252.98	254.50	271.78
	13	204.40	202.77	220.78	227.69	227.31	270.37
	20	201.57	199.42	224.98	224.39	223.90	286.82
0.12	5	247.70	242.12	249.71	262.31	264.60	283.45
	8	226.93	219.75	228.84	244.85	244.44	271.78
	13	213.16	206.04	220.78	227.67	226.79	270.37
	20	211.94	204.80	249.71	225.44	224.91	286.82

time of both ESS and WCS is less when compared to pureDLT strategy. This is due to the fact that the waiting time for the last PTC to arrive penalizes the performance significantly. Also, for a given η, as we increase the network size, the processing time decreases. However, The difference in the processing time between two different values of η for pureDLT increases dramatically as network size increases, but this difference almost remains the same for ESS and WCS. Thus each of the strategies seem to be robust in behavior with respect to the variation of η and n. This behavior can also be observed for different r_f values.

When comparing the performance of ESS and WCS with respect to η, we find that when η is small, WCS shows a better performance than ESS, however, when η is large, the performance of ESS and WCS are almost the same. This is because as η grows, the range of PTC responses (i.e., the time difference between the last PTC and the first PTC) will increase correspondingly, which will naturally benefit ESS, as WCS has to wait longer to start computing.

Effect of ε : Parameter ε determines the size of L_1 and here, ESS and WCS are our only concern. As shown in Tables 1 and 2, both ESS and WCS have a better performance for a smaller ε. This is because that a smaller L_1 implies a shorter $phase_1$; and hence, those processors responded during $phase_1$ will start their computation earlier than for a larger L_1 choice. We also observe that the influence of ε to WCS is less significant than to ESS. This is because WCS has more computation power in $phase_1$ than ESS. Therefore, WCS shows a significantly better performance than ESS with a larger ε.

In fact, the sizes of $L_1, L_2, ...L_k$ all influence the performance of ESS and WCS, but choice of L_1 plays a crucial role. In general, if we partition the load into several smaller portions, the performance of ESS and WCS will increase dramatically, and will be close to the lower bound where computation and communication speeds parameters are known in advance. However, in practice one cannot partition the load into infinitesimally small size portions for processing.

An inverse effect of ε can be observed when both r_f and n are small, and η is big. In this case, as opposed to a common expectation that the finish time would increase as we tend to increase ε, our results show that the finish time decreases. An explanation to this anomalous behavior may be explained as follows. When r_f and n are small, there are limited number of fast processors in the network. Furthermore, as η is large, the range of PTC response is also large. In this situation, a small ε implies that few processors will return their PTCs during $phase_1$, while a larger ε gives more chance for these processors to respond during $phase_1$, and this computation power can be utilized earlier. From this point, we can see that the fact that an unequal partitioning of load among the phases may also under-utilize processors. For instance, a smaller choice of L_1 and a larger choice of L_2 may have an adverse effect, as there may be fewer processors returning their PTCs during $phase_1$, and hence, very few processors will be engaged in $phase_2$.

Effect of r_f : The ratio of fast processors in network is important in determining the performance of ESS and WCS. As we can see from Tables 1 and 2, when r_f increases, overall finish time decreases, which is expected. Further, comparing the performances of ESS and WCS with respect to r_f, we find that WCS prefers a network with a large ratio of fast processors. This is because the fact that when a network has a large number of fast processors, there is higher chance that more fast processors will return their PTCs immediately after the first response. In ESS, these fast processors will wait until the end of $phase_1$ to be engaged in computation, while in WCS, this fast processor will be used much more earlier. Thus, when $r_f = 0.75$, we find WCS shows a significant improvement than ESS. However, when a network has only a few fast processors, WCS may have to wait more time for the response from another processor than ESS. Therefore, when $r_f = 0.25$, WCS and ESS exhibit approximately the same performance.

Effect of network size (n) : Network size is crucial to handle large scale data processing. From Tables 1 and 2, we observe that as the network size increases, the finish time for both ESS and WCS decreases. However, this may not be the case for pureDLT. After n grows to some extent, increasing n further will have an adverse effect on pureDLT, as its finish time starts to increase rather than decreasing. This is because the range of PTCs stretches as n increases, which penalizes pureDLT for its "greedy" waiting nature. Also, notice that when the network size becomes large, the performance of ESS and WCS saturates. This is because that the processors residing at the end of the long chain actually receive very little amount of load due to the cumulative communication delay.

In the above experiments, WCS only waits for two processors before starting $phase_1$. Actually, WCS can be allowed to wait for more processors. We use WCS(k) to denote the algorithm that will wait on k processors before starting $phase_1$. Thus, ESS is WCS(1). For a given network, with fixed ε and η, the finish time obtained in computing L is a function of k (we denote this finish time as $T_{WCS}(k)$). The performances for different k with respect to ε and η can be seen from Figures 2 and 3, for different r_f distribution.

In general, for a given η and ε value, as k increases, $T_{WCS}(k)$ is observed to decline first, reaching a minimum point (shown as point A in the figures) and then increases (See Figure 2 and 3). However, as we decrease η, the entire curve $T_{WCS}(k)$ shifts down, which means less finish time is achieved. Further,

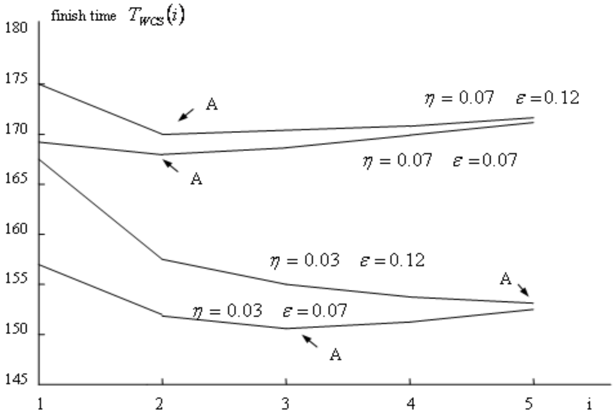


Fig. 2. Figure of $T_{WCS}(i)$ for Different ε and η when $n = 15$ $r_f = 0.75$

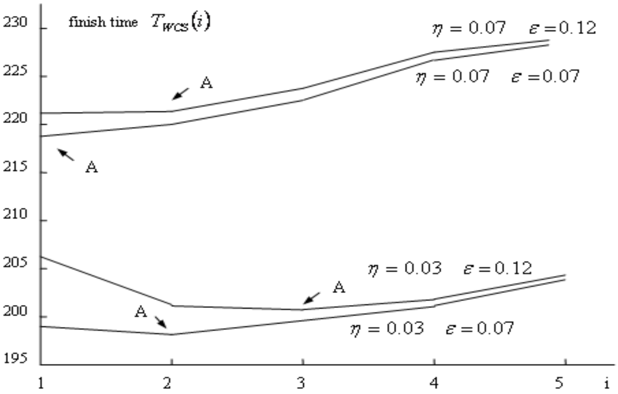


Fig. 3. Figure of $T_{WCS}(i)$ for Different ε and η when $n = 15$ $r_f = 0.25$

the minimum point A starts to shift to right, which means that minimum finish time will be obtained for larger k values. This observation could be useful while implementing the strategies. Thus, if we choose a small η, it would be appropriate to choose a relatively larger k, say 3 or more, to minimize finish time. L_1 is another factor that affects the shape of the curve $T_{WCS}(k)$. When L_1 decreases, the curve $T_{WCS}(k)$ shifts down, and its minimum point shifts to left, which means minimum finish time is obtained for a small k. Furthermore, as shown in Figures 1 and 2, r_f also affects $T_{WCS}(k)$. Larger r_f naturally benefits WCS as discussed above, which drives the minimum point of $T_{WCS}(k)$ to right.

As we have seen above, the value of k which achieves the minimum finish time (point A in the figures) is affected by the combined effect of η, ε, r_f and n, where η and ε are determined by the strategy, while r_f and n are characteristics of the network. n is usually a known parameter. Hence, if we have some prior knowledge about r_f, we can choose a suitable k according to the value of η and ε. The following simulations reveal the most probable best value of k with respect to certain η, ε, r_f and n.

For a network with 15 processors, we first set $\eta = 0.03$ and $\varepsilon = 0.12$, and vary the value of r_f to 0.2, 0.5, and 0.8, respectively. Each category of experiments is repeated 25 times. We find that when $r_f = 0.2$, most of the times (19/25) the minimum finish time is obtained at $k = 2$. When r_f increase to 0.5, to obtain the minimum finish time, k should increase to 3 (17/25), and when $r_f = 0.8$, most of the times the minimum finish time is obtained at $i = 4$ (13/25) or $k = 5$ (9/25). Then we adjust η to 0.07, ϵ to 0.07 and redo the experiment. We find when $r_f = 0.2$, the best value of k equals to 1 (15/25), and when $r_f = 0.5$ or $r_f = 0.8$, most of times (around 21/25) $k = 2$ is the best choice.

5 Conclusions

In this paper we have presented a novel strategy WCS, for scheduling and processing a divisible load on resource unaware linear networks. Since the underlying network is a linear chain of processors, the choice of including the processors for computation becomes crucial in deciding the overall performance of the strategy, as the processor and link speeds are not known a priori. Any inadvertent choice of processors, may slow down the computation. WCS takes distinct advantage in utilizing the first k responded processors earlier in the computation and progressively including other fast processors as time progresses. This special design is actually a general form of an early proposed strategy ESS. We have compared their performance both analytically and through simulation studies. We have also analyzed the performance of a strategy that serves as an upper bound (pureDLT). Our simulation reveals that although in some rare case ESS may have better performance, $k \geq 2$ seems to be a wiser choice in most cases. Notice that a more dedicated network model is considered in this paper. However, to incorporate the long-term perturbations of processor and link speeds, one may attempt to probe during each phase to estimate the speeds dynamically. In this case, strategies may incur large overheads and implementation becomes more complex.

References

1. Cheng, Y.C., Robertazzi, T.G.: Distributed Computation with Communication Delays. IEEE Transactions on Aerospace and Electronic Systems 24, 700–712 (1988)
2. Bharadwaj, V., Ghose, D., Mani, V., Robertazzi, T.G.: Scheduling Divisible Loads in Parallel and Distributed Systems. IEEE Computer Society Press, Los Alamitos (1996)
3. Robertazzi, T.: Ten Reasons to Use Divisible Load Theory. Computer 36(5) (2003)
4. Barlas, G., Veeravalli, B.: Optimized Distributed Delivery of Continuous-Media Documents over Unreliable Communication Links. IEEE Transactions on Parallel and Distributed Systems 16(10), 982–994 (2005)
5. Blazewicz, J., Drozdowski, M., Markiewicz, M.: Divisible Task Scheduling - Concept and Verification. Parallel Computing 25, 87–98 (1999)
6. Ghose, D., Kim, H.J.: Computing BLAS level-2 operations on workstation clusters using the divisible load paradigm. Mathematical and Computer Modelling 41 (2005)
7. Drozdowski, M., Wolniewicz, P.: Performance Limits of Divisible Load Processing in Systems with Limited Communication Buffers. Journal of Parallel and Distributed Computing 64(8), 960–973 (2004)
8. Wolniewicz, P.: Multi-installment Divisible Job Processing with Communication Startup Cost. Foundations of Computing and Decision Sciences 27 (2002)
9. Yang, Y., van der Raadt, K., Casanova, H.: Multiround Algorithms for Scheduling Divisible Loads. IEEE Trans on Parallel and Distributed Systems 16(11), 1092–1102 (2005)
10. Sohn, J., Robertazzi, T.G.: Optimal Time-varying Load Sharing for Divisible Loads. IEEE Transactions on Aerospace and Electronic Systems 34 (1998)
11. Ghose, D., Kim, H.J., Kim, T.H.: Adaptive Divisible Load Scheduling Strategies for Workstation Clusters with Unknown Network Resources. IEEE Transactions on Parallel and Distributed Systems 16(10), 897–907 (2005)
12. Jingxi, J., Bharadwaj, V.: Resource Unaware Computing - A Distributed Strategy for Divisible Load Processing on Linear Daisy Chain Networks. In: The 14th IEEE International Conference on Networks (ICON 2006), Singapore (2006)
13. Hung, J.T., Robertazzi, T.G.: Divisible Load Cut Through Switching in Sequential Tree Networks. IEEE Transactions on Aerospace and Electronic Systems 40(3), 968–982 (2004)

A Novel Learning Based Solution for Efficient Data Transport in Heterogeneous Wireless Networks

B. Venkata Ramana, K. Srinivasa Pavan, and C. Siva Ram Murthy

Department of Computer Science and Engineering, IIT Madras, India - 600036
{vramana,pavan}@cse.iitm.ernet.in, murthy@iitm.ac.in

Abstract. There has been a spectacular growth in the use of wireless networks in recent times and consequently, adapting TCP to the wireless networks is a hot topic of current research. However, most of the existing works proposed for this problem have been designed for specific wireless networks, or they necessitate changes at either the receiver, at the intermediate nodes, or at both, because of which their deployment become very difficult. Therefore, here we propose a TCP variant which works over both multi-hop ad hoc wireless networks as well as single-hop wireless networks. We use a learning based method to dynamically change the congestion window size according to the network conditions. Our protocol does not rely on any explicit feedback from the network and requires only sender-side modifications. Through extensive simulations we show that our protocol achieves the performance improvement, in terms of goodput, packet loss, and fairness to the competing flows.

1 Introduction

Transport Control Protocol (TCP) dominates today's communication in various networks. While responding to congestion in the network, it offers reliable data transport service to the applications. To prevent buffer overflow at the routers, it controls its rate of transmission by maintaining a congestion window (cwnd), which is an upper bound on the maximum number of unacknowledged packets in the network. TCP adjusts its cwnd in a deterministic fashion according to the network events. Refer [1] for more details about TCP.

1.1 A Brief Introduction to Various Wireless Networks

Ad hoc wireless networks are multi-hop networks in which the nodes use multi-hop relaying to communicate with the nodes that are not directly reachable. They have limited bandwidths and high wireless losses. These networks typically have a low bandwidth-delay product (BDP). *Satellite links* are characterized by high wireless losses, latency, and bandwidth. These networks have high BDP and link asymmetry, where the bandwidth on downlink is usually much higher than that on uplink. *Cellular networks* have moderately high bandwidth and

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 402–414, 2008.

high latencies. They have moderate wireless losses and link asymmetry, and they often experience bandwidth oscillations and high delay variations due to handoffs. *Wireless LANs* have low latencies and high bandwidth. However, due to the link level retransmission scheme to handle the wireless losses, WLANs also have delay fluctuations. In WLANs, uplink and downlink are same and both compete with each other for shared bandwidth.

1.2 Limitations of TCP in Wireless Networks

There is a huge growth in the use of wireless networks in recent times and consequently, adapting TCP to the wireless networks is a hot research topic. The following are some of the problems faced by TCP in wireless networks ([2]).

Wireless losses: Congestion is the main cause of packet loss in wired networks. However, in wireless networks packet loss could also happen due to erroneous wireless links or user mobility. TCP assumes these losses to be congestion losses and reduces its cwnd, which adversely affects the throughput. *Bandwidth-delay product:* Satellite links have high BDP. So, TCP should increase its cwnd aggressively. In contrast, as ad hoc networks have low BDP, TCP should follow a conservative approach to increase its cwnd. However, in congestion avoidance (CA) phase, TCP increases its cwnd by $\frac{1}{cwnd}$ for every TCP acknowledge (ack) it receives, which is too small in satellite networks and too large in ad hoc networks. Hence, TCP should dynamically adjust its increment factor depending on the network in which it operates. *Reactive nature:* TCP keeps increasing its cwnd until it experiences a packet loss. Thus, it is reactive rather than proactive in avoiding losses. TCP needs to resend these lost packets which not only reduces the throughput, but also severely affects the network resources.

1.3 Goals

We aim to design a TCP variant which works in both the single-hop wireless networks (WLANs, cellular, and satellite networks) as well as in multi-hop ad hoc networks, while attaining the following goals.

- Throughput improvement while reducing the packet losses.
- No explicit feedback from any of the *network components* (i.e., the receiver node or the intermediate nodes). Because in reality, one cannot expect changes at all the network components.
- Fairness among competing flows.

2 Related Work

Extensive research has been done to adapt TCP to address the challenges in the wireless domain. However, most of the work was done for specific wireless network paradigms. For example, the proposals TCP-Peach+ [3] (for satellite networks), WTCP [4] (cellular networks), Snoop [5] (WLANs), and LTCP [6] (ad hoc networks) are designed for a specific wireless network.

The proposals [7]-[9] are designed for more than one kind of network. TCP-NewJersey [7] was shown to perform well in WLANs and cellular networks. However, it relies on explicit congestion warnings from the network. TCP-Westwood [8] claims improved performance over TCP in WLANs and LEO satellite networks, while achieving fairness. However, it dependents on the acks because of which its performance degrades when it is used in high delay links. This is due to the poorer accuracy in the bandwidth estimation done based on the delayed acks [9]. Adaptive Transport Layer (ATL) [9] has been designed for the single-hop wireless networks, which uses the TCP Friendly Rate Control equation to estimate the throughput of the TCP flow in the wired part of the network, and varies its additive increase parameter α to attain a similar throughput. We compare our work with ATL as it was claimed to work in WLANs, satellite, and cellular networks and had a significant performance gain over Snoop, WTCP, Peach+, and Westwood. However, ATL has several limitations which make its deployment extremely difficult. (a) ATL needs the receiver to explicitly supply the wireless link loss and delay values at the receiver's side. (b) ATL tries to attain the throughput of the wired part of the network by increasing its α aggressively. This causes an aggressive increase in cwnd and consequent unfairness to the competing flows in the wireless part of the network. (c) ATL was not designed for ad hoc networks. (d) ATL is a reactive protocol.

3 Overview of Our Protocol

TCP increments its cwnd by a deterministic value. In CA phase, it increases its cwnd by 1 MSS (Maximum Segment Size) for every RTT (Round-Trip Time). This is a big drawback and limits its extensibility to various networks. For instance, satellite networks with high BDP require a larger cwnd increment to quickly utilize the large bandwidth available, which is in contrast to ad hoc networks. Also because of its reactive nature, TCP experiences high packet loss, which leads to a poor utilization of network resources. To overcome these problems, we propose a learning-based mechanism, which uses Finite Action Learning Automata (FALA), to dynamically update the cwnd based on the network conditions. The idea behind FALA is that there is a set of actions and a set of action probabilities for choosing these actions. These probabilities are updated in such a way that the favorable actions have a higher probability of being chosen. In our case, the actions are the absolute increment (or decrement) in the cwnd.

The advantages of our protocol (Learning-TCP) are the following. Learning-TCP has a wide range of actions, because of which it can increase its cwnd aggressively or conservatively, based on the network it is operating in. In addition, as the actions are learnt from the network conditions, it becomes a proactive protocol. Hence, when incipient congestion is detected, it reduces cwnd and reduces the congestion-related losses. This proactive decrease in cwnd not only reduces the packet loss, but also leads to better fairness to the competing flows.

The following steps, also shown in Fig. 1, give a brief description of the protocol. They are explained in detail in Section 4.

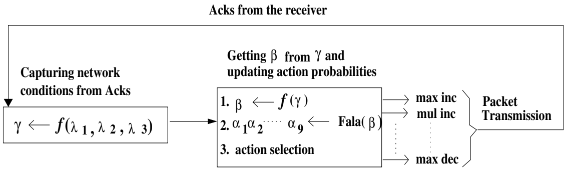

Acks from the receiver
Capturing network conditions from Acks
γ ← f(λ 1, λ 2, λ 3)
Getting β from γ and updating action probabilities
1. β ← f(γ)
2. α 1α 2 ···· α 9 ← Fala(β)
3. action selection
max inc
mul inc
max dec
Packet Transmission

Fig. 1. Learning-TCP – inferring network conditions and selecting appropriate action

1. *Get the network conditions:* Using the RTTs and the forward path throughput, we capture the congestion and throughput fluctuations in the forward path into the parameters λ_1 and λ_2, respectively. Further, in order to reduce the number of timeouts, using the cwnd values at which previous timeouts have happened, we compute the cwnd at which the next timeout is likely to happen. Taking this as a reference point, we obtain the degree of aggressiveness needed in cwnd increase as a parameter λ_3.
2. Map λ_1, λ_2, and λ_3 to a single parameter γ, which corresponds to the network response or network feedback.
3. Map network feedback γ to β (input to FALA) and using which update the action probabilities.
4. Select an action stochastically and update the cwnd accordingly.

4 Protocol Details

A learning automata system [10] consists of a finite set of states $(\phi_1, \phi_2, \ldots, \phi_n)$ and a finite set of actions corresponding to these states $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, hence called as Finite Action-set Learning Automata (FALA), which can be operated in a random environment. At a time instant n, based on the previous actions and their responses from the environment, the FALA selects an action from the action set. Then, it may receive either favorable or unfavorable feedback from the environment by the next time instant $n+1$; using a transition function and taking feedback into account, FALA will decide on the new state $\phi(n+1)$ and performs the corresponding action of this new state. The transition function is such that the FALA will move to states where the favorable actions are chosen more often. The advantages of learning automata are (a) The learning equations are fairly simple and their convergence proofs are well established. (b) It does not require any modeling of the environment. (c) It works in any random environment and dynamically learns by observing the environment conditions.

We use FALA to learn and probabilistically decide on the amount of absolute change in the cwnd. In our case, we have nine actions $(\alpha_1, \alpha_2, \ldots, \alpha_9)$ which correspond to the amount of increase or decrease in cwnd when the respective action is performed. The actions, their corresponding increase or decrease in cwnd, and the values of the network feedback γ at which these actions are favored are given in Table 1. For an action performed at time instant n, $\alpha(n)$, we obtain the input to the FALA, $\beta(n)$, from the network feedback γ.

Table 1. Actions and their effective changes in cwnd size in FALA

Action $(\alpha_1, \alpha_2, \ldots, \alpha_9)$	**Change in** **cwnd size**	**Favorable** **value of** γ	**Calculation** **of** β **from** γ
max_inc	cwnd += 2*MSS	1	$1 - \gamma$
mul_inc	cwnd += MSS	0.9	$\frac{\lvert 0.9-\gamma \rvert}{0.9}$
add_inc	cwnd += $\frac{1}{cwnd}$	0.7	$\frac{\lvert 0.7-\gamma \rvert}{0.7}$
min_inc	cwnd += $\frac{0.01}{cwnd}$	0.6	$\frac{\lvert 0.6-\gamma \rvert}{0.6}$
no_chng	cwnd = cwnd	0.5	$\frac{\lvert 0.5-\gamma \rvert}{0.5}$
min_dec	cwnd -= $\frac{0.01}{cwnd}$	0.4	$\frac{\lvert 0.4-\gamma \rvert}{0.6}$
add_dec	cwnd -= $\frac{1}{cwnd}$	0.3	$\frac{\lvert 0.3-\gamma \rvert}{0.7}$
mul_dec	cwnd -= MSS	0.1	$\frac{\lvert 0.1-\gamma \rvert}{0.9}$
max_dec	cwnd -= 2*MSS	0	γ

4.1 Inferring Network Conditions

As we do not rely on any explicit support from the network, we infer the network conditions (as we discuss in this section) from the acks which are our only source of knowledge about the network.

Mapping of Congestion: RTT is potentially a good metric to estimate the congestion in the network. In order to estimate congestion, we keep the mean and standard deviation (sdev) of most recent n RTTs. As we can observe from Fig. 2, in the presence of congestion there is a sharp increase in the RTTs. Further, sdev increases with the increasing level of congestion. Hence, we can detect the incipient congestion, from the sdev of RTTs. The RTTs which are above $mean + sdev$ indicate a sign of congestion in the network. However, in order to reduce the chances of wireless fluctuations from being falsely identified as congestion, we take the upper bound for indicating congestion as $mean{+}2{\times}sdev$. Since the RTTs below $mean - sdev$ indicate the availability of free bandwidth in the network, we take the lower bound for RTTs as $mean - sdev$.

On a linear scale between 0 and 1, with the normalized values of $mean + 2 \times sdev$ and $mean - sdev$ as the upper and lower bounds, respectively, the mean of k recent RTTs ($k < n$), is mapped to a parameter λ_1 (see Fig. 4.a). We consider mean of k recent RTTs ($mean_k$) instead of only the current RTT. This is to deal with the fluctuations in RTTs which are caused by the bandwidth fluctuations and link-level error control. Any value below $mean - sdev$ is mapped to 1, to indicate absence of congestion in the network and any value above $mean{+}2{\times}sdev$ is mapped to 0 to indicate that the network is congested. We empirically found the values for n and k as 100 and 15, respectively. These values allow us to consider reasonable amount of history for making decisions about congestion.

Mapping of Throughput: Throughput in the forward path is another important parameter in wireless networks because of asymmetric paths. From the time stamp in the ack, the sender infers the arrival time of data packet at the receiver. To avoid the fluctuations in the estimation, the sender computes the throughput over recent k acks. Fig. 3 shows the trends in forward path throughput.

To map the throughput to λ_2, we keep the maximum throughput ($tput_{max}$) observed so far. However, taking just the $tput_{max}$ will not completely indicate

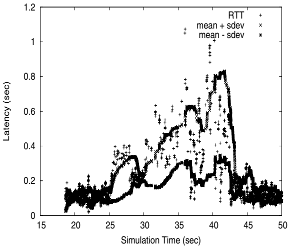


Fig. 2. RTTs of a TCP flow along with the $mean+sdev$, and $mean - sdev$ of RTTs, in the presence of background traffic between the time 25 secs and 40 secs

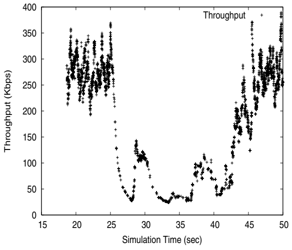


Fig. 3. The corresponding forward path throughput which drops drastically when background traffic exists and this drop persists until the congestion ends

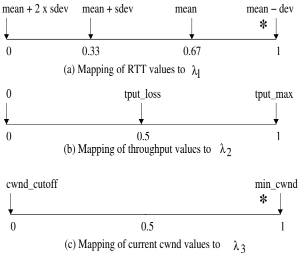

mean + 2 x sdev
mean + sdev
mean
mean – dev
*
0
0.33
0.67
1
(a) Mapping of RTT values to λ1
0
tput_loss
tput_max
0
0.5
1
(b) Mapping of throughput values to λ2
cwnd_cutoff
min_cwnd
*
0
0.5
1
(c) Mapping of current cwnd values to λ3

Fig. 4. Mapping of network conditions to λ_1, λ_2, λ_3. Note that $*$ denotes the favorable bound of λ_1 and λ_3 for cwnd increase action; for λ_2, the bound varies with network conditions.

the current network conditions as one best case $tput_{max}$ will always try to increase the cwnd to achieve this value. Hence, we introduce the throughput at the previous loss ($tput_{loss}$) as a second reference point, at the mid scale. In the new mapping, the values between 0 and $tput_{loss}$ are mapped to [0,0.5] and the values between $tput_{loss}$ and $tput_{max}$ are mapped to [0.5,1] (see Fig. 4.b). Hence, when the throughput is very low, (1-λ_2) will be close to 1 and we favor increase actions. When the throughput is close to $tput_{loss}$, we favor *no_chng* action, and when the throughput is close to $tput_{max}$, we favor decrease actions.

Avoiding Timeouts: As Learning-TCP may increase the cwnd by more than 1 MSS in the CA phase, there could be successive timeouts due to multiple losses. To avoid this, we bring cwnd into feedback. We maintain the values for parameters $cwnd_{prev}$ and $cwnd_{mean}$, which represent the cwnd at the previous timeout and the mean of the cwnd sizes at all previous timeouts, respectively. At each ack arrival, using an auto-regressive technique shown in Eq. 1, we obtain the cwnd at which the next timeout is likely to occur ($cwnd_{cutoff}$).

$$cwnd_{cutoff} = f(t) \times cwnd_{prev} + (1 - f(t)) \times cwnd_{mean} \tag{1}$$

Here $0 < f(t) < 1$ and $f(t)$ is a monotonically decreasing function with time t. The idea is that initially $cwnd_{cutoff}$ will be close to the $cwnd_{prev}$. However, if there is no timeout for a long time, $cwnd_{prev}$ may become a bottleneck. Hence, as time progresses, $cwnd_{cutoff}$ is gradually shifted towards the average cwnd at which most of the timeouts happened. Using $cwnd_{cutoff}$ as a reference point, we capture the degree of aggressiveness needed in cwnd increase into λ_3. On a linear scale between 0 and 1, with the normalized values of $cwnd_{cutoff}$ and minimum cwnd as the upper and lower bounds, respectively, the current cwnd is mapped to λ_3 (see Fig. 4.c). Any cwnd size above $cwnd_{cutoff}$ is mapped to 0.

4.2 Mapping Network Conditions into a Single Parameter γ

Here, we discuss how to combine λ_1, λ_2, and λ_3 into a parameter γ, which provides the network feedback. A high γ indicates a favorable sign for cwnd increase and a low γ indicates a favorable sign for cwnd decrease. The first three cases shown below combine λ_1 and λ_2 into γ, then the fourth case combines λ_3 and γ.

1. We detect the congestion if $mean_k > mean + sdev$ ($\equiv \lambda_1 < 0.33$), in which case throughput will also be low. In this case, we determine the degree of intensity needed in decreasing cwnd size, by taking $\gamma =$ min $\{\lambda_1, \lambda_2\}$. Depending on γ the FALA favors either *max_dec*, *add_dec*, or *min_dec*.
2. When the $mean_k$ is around or lower than $mean$ ($\equiv \lambda_1 > 0.66$), the network is probably not congested hence, the feedback should favor the cwnd increase actions. In this case, we take $\gamma =$ max $\{\lambda_1, 1 - \lambda_2\}$. This case is very useful in quickly increasing the cwnd after congestion. Just after congestion, the RTTs fall and hence, λ_1 will be high. Besides, just after congestion the throughput will be low and so $1 - \lambda_2$ will be high indicating that there is a lot of unused bandwidth in the network. We take the maximum of these two values as a feedback to the FALA to quickly increase the cwnd.
3. For the remaining values of $mean_k$ (i.e., $mean + sdev < mean_k < mean$), we take $\gamma = \frac{\lambda_1 + \lambda_2}{2}$, which gives equal importance to both the RTT and throughput parameters. For instance, when both λ_1 and λ_2 are high, a high value of λ_1 indicates that the network is not congested. However, λ_2 is high implies that we are close to the maximum throughput and so we should be cautious, and $\gamma = \frac{\lambda_1 + \lambda_2}{2}$ takes both these factors into account.
4. Then γ is adjusted as $\gamma = \frac{\gamma + \lambda_3}{2}$. When λ_3 is close to 1, γ will be adjusted between 1 and 0.5 and thus increase actions will be further favored. When λ_3 is close to 0.5, γ will be mapped between 0.75 and 0.25, and consequently additive increase/decrease actions will be favored. Finally, when λ_3 is close to 0, γ will be mapped between 0.5 and 0, and decrease actions will be favored.

4.3 Learning Algorithm and Action Selection

After mapping the network conditions to γ, using the equations in Table 1, we map γ to β, which forms an input to FALA. The idea behind the mapping is that for a favorable feedback (i.e., reward), β expects a value close to 0, so that the probability for the action selected at previous time step will be increased, making it a more favorable, and accordingly other action probabilities will be decreased. For unfavorable feedback (i.e., penalty) it reacts just oppositely. For instance, when *max_inc* action is performed and if the feedback is favorable $(i.e., \gamma \equiv 1)$ then the corresponding equation $1 - \gamma$ in the table assigns β a value close to 0, which further increases the probability for selecting *max_inc* action.

Due to the continuous nature of β and the possibility of multiple actions, we use the S-model reinforcement scheme with multi-action linear reward-penalty scheme L_{R-P}, known as SL_{R-P} for updating the action probabilities [10]. In the SL_{R-P} scheme, responses may be partly favorable and partly unfavorable to the action. The extent of favorability is specified as a value between 0 and 1. Eq. 2 gives equations for SL_{R-P} reinforcement scheme.

$$\begin{aligned}p_i(n+1) &= p_i(n) - (1-\beta(n))g_i(p(n)) + \beta(n)h_i(p(n)) \quad if\ \alpha(n) \neq \alpha_i \\ p_i(n+1) &= p_i(n) + (1-\beta(n))\textstyle\sum_{j\neq i} g_j(p(n)) - \beta(n)\sum_{j\neq i} h_j(p(n)), if\ \alpha(n) = \alpha_i\end{aligned} \tag{2}$$

The action probabilities $p_i(n+1)$ at time step $n+1$ are computed from action probabilities $p_i(n)$ at time step n and the current input values of $\beta(n)$. The α_i, in the equation, refers to the action selected at time step n. The functions $g_j(.)$ and $h_j(.)$ correspond to reward and penalty functions of the multi-action L_{R-P} scheme, respectively and are computed based on Eq. 3.

$$\begin{aligned}g_j(p) &= ap_j(n) \\ h_j(n) &= \frac{a}{r-1} - ap_j(n)\end{aligned} \tag{3}$$

The terms a and r in these equations correspond to the *learning factor* and *number of actions*, respectively. In order to maintain convergence property, we take a as 0.01, which is a small value.

Action selection is done stochastically. To select an action, FALA generates a uniform random number and then using the updated action probabilities, it determines the probability that closely matches the random number and the corresponding action is selected. This feature helps FALA to perform various actions efficiently and accordingly adapt to changing network conditions. Since the preferred actions are assigned higher probabilities, there is a high probability for these actions to be selected, and at the same time the actions with lower probabilities are not completely ignored.

5 Simulation Results

We carry out our simulations over *ns*-2.28. For the single-hop wireless networks (WLAN, cellular, and satellite networks) we take a dumbbell topology. Over a wireless link, the senders are connected to a router R1, which in turn connected to Router R2 with a wired link; all the receivers are connected over wired links to R2. The wired links have 10 Mbps bandwidth and 10 ms delay. The results produced in this section are obtained with 95% confidence level. In all the sumulations, we discard the first 2000 packets and then collect statistics over the next 500 secs. We use FTP traffic and UDP flows as background flows, where each UDP flow sends 100 pkts/sec. We compare our Learning-TCP against TCP NewReno and Sack enabled ATL [9] for the following performance metrics.

1. **Goodput** is the number of bytes successfully transmitted per unit time.
2. **Loss Percentage or Packet Loss** is $\frac{total\ bytes\ lost}{total\ bytes\ transmitted}$.
3. **Bandwidth Stolen** is a measure of inter-protocol fairness. If a NewReno flow has a goodput of T_1 when competing with another NewReno flow and T_2 when competing with an aggressive protocol, then bandwidth stolen by the aggressive one is $(T_1 - T_2)/T_1$ if $T_1 > T_2$, and 0 otherwise.

5.1 Simulations in Ad Hoc Wireless Networks

We now study the performance of Learning-TCP in ad hoc networks and compare it with that of TCP NewReno and LTCP [6]. We perform simulations on chain and grid topologies. One main advantage of Learning-TCP over NewReno in ad hoc networks is its conservative and proactive nature, which should intuitively result in a higher goodput for Learning-TCP with a lower packet loss.

Figs. 5 and 6 show the goodput and packet loss for the three protocols at varying hop lengths. At all hop lengths, Learning-TCP outperforms the other two protocols. Learning-TCP shows upto 62% higher goodput and upto 64% lower packet loss than NewReno, where as LTCP shows upto 30% higher goodput and upto 38% lower packet loss, than NewReno. Next, on an 8x8 grid, leaving the four corner points, we took 6 horizontal and 6 vertical flows of hop length of 7 each. This is an extremely congested network. Fig. 7 shows the goodput attained by each of the 12 flows. Cumulative goodput of NewReno, LTCP, and Learning-TCP are 29, 34.4, and 50.5 Kbps, respectively, and their corresponding loss % are 43.5%, 37.9%, and 24.3%, respectively. LTCP has an 18% improvement in cumulative goodput and 13% reduction in packet loss, than NewReno. Learning-TCP is the best among the three, with a 74% improvement in cumulative goodput and 44% reduction in packet loss than NewReno.

5.2 Simulations in Satellite Networks

Here, we discuss the performance of Learning-TCP over satellite links. We use satellite MAC in *ns*-2.28 and set the uplink and downlink bandwidths as 11 Mbps

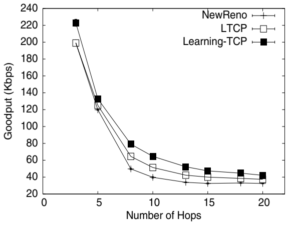


Fig. 5. Goodput vs Hop Length

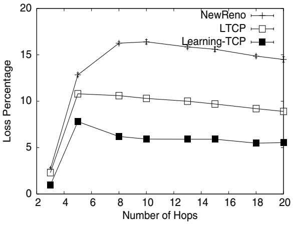


Fig. 6. Loss % vs HopLength

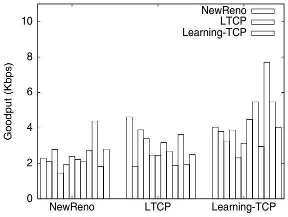


Fig. 7. Goodput in an 8x8 grid ad hoc network

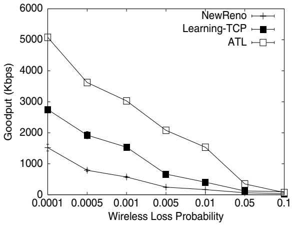


Fig. 8. Goodput in satellite networks for varying losses

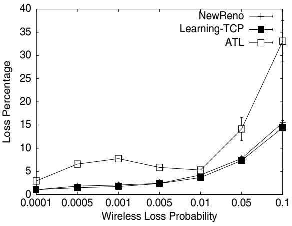


Fig. 9. Loss % in satellite networks for varying losses

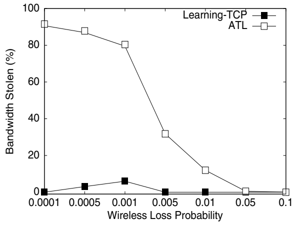


Fig. 10. Bandwidth stolen in satellite networks

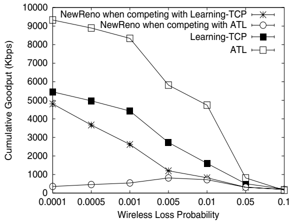


Fig. 11. Cumulative goodput in satellite networks for 5 flows

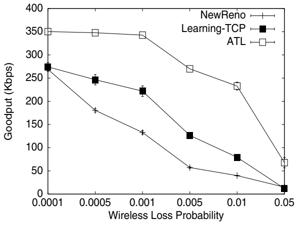


Fig. 12. Goodput in cellular networks for varying losses

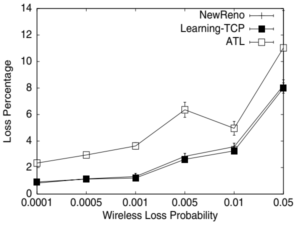


Fig. 13. Loss % in cellular networks for varying losses

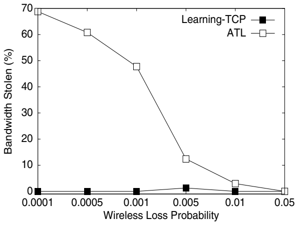


Fig. 14. Bandwidth stolen in cellular networks

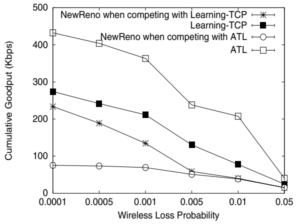


Fig. 15. Cumulative goodput in cellular networks for 5 flows

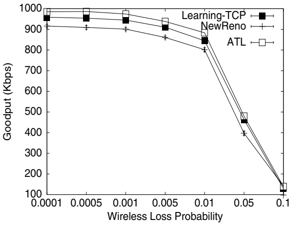


Fig. 16. Goodput in WLANs for varying losses

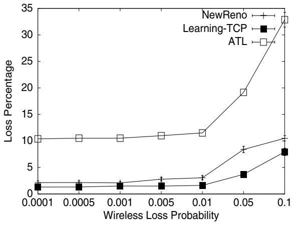


Fig. 17. Loss % in WLANs for varying losses

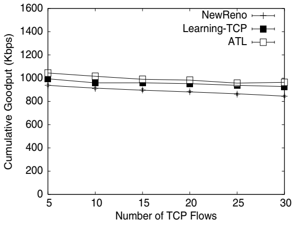


Fig. 18. Cumulative goodput for multiple flows in WLANs

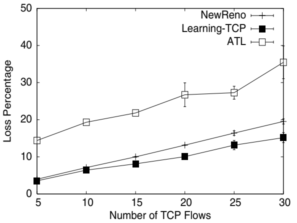


Fig. 19. Mean loss % for multiple flows in WLANs

and 40 Mbps, respectively. The one way link delay is taken as 250 ms. The advantage over NewReno is that Learning-TCP quickly increases its cwnd to use the bandwidth available and thereafter fluctuates around the $cwnd_{cutoff}$. Besides, even on encountering losses, Learning-TCP quickly reaches the $cwnd_{cutoff}$.

First, in the presence of background traffic (9 UDP flows), we study the performance of a flow of Learning-TCP at various loss probabilities. The results are shown in Fig. 8. At all loss probabilities, Learning-TCP shows significantly higher goodput than NewReno, with a peak improvement of 168% at 0.005 loss probability. Because of its aggressive nature, ATL attains significantly higher goodput over Learning-TCP and NewReno. However, soon we show that this improvement is at the cost of a high unfairness to the competing TCP flows and a high packet loss. As we increase the loss probability, wireless losses become dominant, due this the goodputs of these three protocols are decreasing with increasing loss probabilities. Fig. 9 shows the comparison of packet loss for these three protocols. We notice that Learning-TCP shows marginally lower packet loss than NewReno. ATL, on the other hand, has a significantly higher packet loss than both NewReno and Learning-TCP.

Next, we show that the goodput improvement shown by Learning-TCP is not at the cost of unfairness to the competing TCP flows. We first take ten NewReno flows and compute the mean goodput, then replace five of the NewReno flows with five Learning-TCP flows, and subsequently with five ATL flows and observe the bandwidth stolen in both the cases. The comparison of bandwidth stolen for varying loss probabilities is given in Fig. 10. Learning-TCP has significantly better fairness than ATL. The bandwidth stolen by Learning-TCP is nil in most of the cases. On the other hand, ATL increases its cwnd aggressively, causes congestion losses, and forces the competing flows to half their cwnd even before they get their fair share of the bottleneck bandwidth. This is clearly visible at low loss probabilities where congestion losses are dominant. For instance, ATL steals 91.5% of the NewReno bandwidth at a loss probability of 10^{-4}, which is certainly not acceptable. Fig. 11 shows the cumulative goodput attained by both Learning-TCP and NewReno when we have five flows of each type competing together. At all loss probabilities, Learning-TCP has a significant improvement in cumulative goodput upto 120% over NewReno. Also, we can notice that ATL is achieving higher cumulative goodput by causing severe unfairness to the competing NewReno flows. As a result of this the NewReno flows are almost starved by losing upto 91.5% of their bandwidth.

5.3 Simulations in Cellular Networks

We discuss the performance of Learning-TCP over cellular networks. As shown in [2], we use a wired link to simulate cellular link by setting uplink delays and bandwidth as 100 ms and 500 Kbps, respectively; downlink delays and bandwidths are taken as 100 ms and 1 Mbps, respectively. UDP flows generate traffic at 50 pkts/sec; and all other details are same those for satellite network studies.

Figs. 12 and 13 show the goodput and packet loss for varying wireless loss probability. The trends are similar to those of the corresponding studies in satellite networks. Learning-TCP shows significantly higher goodput than NewReno, with a peak improvement of 136% at 0.005 loss probability and about 76% at the typical cellular wireless loss probability of 10^{-3}. ATL shows higher goodput over the other two protocols, however it has higher packet loss than both NewReno and Learning-TCP. Figs. 14 and 15 show the bandwidth stolen and cumulative goodput results, respectively. Learning-TCP shows better fairness to the competing NewReno flows than ATL. The bandwidth stolen by Learning-TCP is almost nil. ATL on the other hand steals upto 68.8% from NewReno.

5.4 Simulations in Wireless LANs

Here, we study the performance of Learning-TCP over WLANs by using IEEE 802.11 MAC operating at 2 Mbps. The other simulation details are same as those of satellite networks.

Figs. 16 and 17 show the goodput and packet loss, respectively, for varying wireless loss probabilities. Learning-TCP shows upto 25% higher goodput over NewReno. ATL shows higher goodput among these, and has extremly a high packet loss over NewReno and Learning-TCP, whereas NewReno shows higher losses over Learning-TCP. Further, we carried out a study by increasing the number of TCP flows at a typical WLAN loss probability of 10^{-4}. Figs. 18 and 19 show the cumulative goodput and packet loss, respectively. We can observe that while showing lower packet loss, Learning-TCP shows consistently higher goodput over NewReno which is clearly visible at higher number of flows.

6 Conclusions

In this work, we proposed a unified reliable data transport protocol (Learning-TCP) for heterogeneous wireless networks. Learning-TCP seeks neither explicit network support nor changes at *network components*, thus enabling its easier deployment. It can operate aggressively in high BDP networks and conservatively in low BDP networks. When the incipient congestion is detected, it reduces cwnd proactively and avoids multiple losses and timeouts, thereby conserving battery power and bandwidth. Through extensive simulations, we showed that Learning-TCP provides higher goodput over NewReno while reducing the packet losses across all the wireless networks. We also compared Learning-TCP against ATL and observed higher goodput for ATL. However, this improvement came at the cost of high packet loss and unfairness to the competing flows which are certainly unacceptable. Learning-TCP improves the throughput in a fair manner without the need for explicit network support thereby achieving the desired goals.

References

1. Tian, Y., et al.: TCP In Wireless Environments: Problems and Solutions. IEEE Communications Magazine 43(3), S27–S32 (2005)
2. Gurtov, A., Floyd, S.: Modeling Wireless Links for Transport Protocols. In: ACM Sigcomm Computer Communication Review, April 2004, pp. 85–96 (2004)
3. Akyildiz, I.F., et al.: TCP-Peach+: Enhancement of TCP-Peach for Satellite IP Networks. IEEE Commun. Lett. 6(7), 303–305 (2002)
4. Sinha, P., et al.: WTCP: A Reliable Transport Protocol for Wireless Wide-Area Networks. In: Proc. ACM Mobicom, August 1999, pp. 231–241 (1999)
5. Balakrishnan, H., et al.: Improving TCP/IP Performance over Wireless Networks. In: Proc. ACM Mobicom, pp. 2–11 (1995)
6. Venkata Ramana, B., Manoj, B.S.: Learning-TCP: A Novel Learning Automata based Reliable Transport Protocol for Ad hoc Wireless Networks. In: Proc. IEEE Broadnets, pp. 521–530 (October 2005)

7. Xu, K., et al.: Improving TCP Performance in Integrated Wireless Communications Networks. In: Proc. IEEE ICDCS, pp. 136–143 (May 1995)
8. Mascolo, S., et al.: TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In: Proc. ACM Mobicom, pp. 287–297 (July 2001)
9. Akan, O.B., Akyildiz, I.F.: ATL: An Adaptive Transport Layer Suite for Next-Generation Wireless Internet. In: IEEE JSAC, vol. 22(5) (June 2004)
10. Narendra, K.S., Thathachar, M.A.L.: Learning Automata: An Introduction. Prentice Hall, New Jersey (1989)

Scalable Data Collection in Sensor Networks

Asad Awan, Suresh Jagannathan, and Ananth Grama

Purdue University, West Lafayette, IN 47907
{awan, suresh, agy}@cs.purdue.edu

Abstract. Dense sensor deployments impose significant constraints on aggregate network data rate and resource utilization. Effective protocols for such data transfers rely on spatio-temporal correlations in sensor data for in-network data compression. The message complexity of these schemes is generally lower bounded by n, for a network with n sensors, since correlation is not collocated with sensing. Consequently, as the number of nodes and network density increase, these protocols become increasingly inefficient. We present here a novel protocol, called SNP, for fine-grained data collection, which requires approximately $O(n - R)$ messages, where R, a measure of redundancy in sensed data generally increases with density. SNP uses spatio-temporal correlations to near-optimally compress data at the source, reducing network traffic and power consumption. We present a comprehensive information theoretic basis for SNP and establish its superior performance in comparison to existing approaches. We support our results with a comprehensive experimental evaluation of the performance of SNP in a real-world sensor network testbed.

1 Introduction

With the goal of building a comprehensive systems infrastructure for scalable sensor networks, we have developed a distributed data-driven processing runtime and an associated flexible programming environment [1] for structural health monitoring. Our experience with this testbed (Bowen Labs), and the experience of other real-world deployments motivate efficient data collection mechanisms from sensor networks. Supporting fine-grained data collection in dense sensornets is rendered challenging by the scarce network and energy resources at sensor nodes. Increasing network density, and hence data rate, results in rapid degradation in wireless neighborhood network capacity [4]. Similarly, increasing data rate results in high energy consumption [3]. Consequently, reducing data traffic is critical to both throughput and longevity of the network. In-network compression using spatio-temporal correlations is a viable application-independent technique for fine-grained data collection.

Several existing protocols (e.g., [2, 8]) exploit spatio-temporal correlations by partitioning the network into disjoint clusters. Data from each node in the cluster is routed to a cluster representative, which then performs correlation driven compression. Compressed data from cluster representatives is then relayed to the sink. The resulting message overhead of such protocols is approximately $O(n \cdot k_h) + O(n_c \cdot k_s)$. Here, n is the number of nodes in the network, k_h is the average number of hops from a source to the cluster head, n_c is the average number of messages with compressed data, and

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 415–426, 2008.

k_s is the average number of hops from the cluster representatives to the sink. The first term in this overhead results from data sharing for correlation and the second term is the actual compressed data. As the density and number of nodes in the network increase, the first term dominates overhead. Furthermore, while such protocols reduce network congestion near the sink, congestion near the sources increases rapidly.

In this paper, we present a novel protocol called spatial neighborhood protocol (SNP) for exploiting spatio-temporal correlations in dense sensornets. The critical differentiating aspect of SNP is that correlation-based compression is collocated with sensing. Each node independently determines whether it should communicate its data based on data received from other nodes in its spatial neighborhood[1]. Based on the assumption that correlations are likely to be spatially localized, a node requires data only from a few other nodes in its neighborhood to (near) optimally compress its data. In this manner, only a subset of the nodes in the network need to broadcast their data. As a result the overhead of SNP is $O(n-R)+O(n_c \cdot k)$. Here, k is the average number of hops from a source to the sink, n_c is the number of messages containing compressed data, and R is a measure of redundancy in the network. The increasing redundancy, R, as a function of node density, is key to the scalability of SNP. We show that SNP achieves near optimal compression without the $O(n)$ overhead for computing correlations.

2 Information Theoretic Underpinnings

In this section we develop the spatial neighborhood (SN) model (Section 2.4), which forms the basis for the SNP protocol. We also present the partitioning model (PT), which forms the basis of existing protocols (Section 2.5). We show that the compression overhead of the SN model is much less than that of the PT model. We also show that the compression rate of the SN model is better than that of the PT model. In demonstrating the near-optimality of compression overhead and rate for the SN model, we establish it as the basis for data gathering protocols in sensornets.

2.1 Preliminaries

Shannon entropy, or simply entropy, of a random variable X, denoted by $\mathcal{H}(X)$ is a measure of the uncertainty (randomness) of a variable. If X is a random variable whose values are drawn from the probability distribution of the data generated by a sensor node, then $\mathcal{H}(X)$ denotes the entropy of the source. To model a network with n nodes, we define $\mathbf{N}$ as a set of random variables. $X_i \in \mathbf{N}$ represents the random variable associated with the data originating at node i and $\mathcal{H}(X_i)$ represents its entropy. Joint entropy of multiple sources corresponds to the minimum amount of information that can be used to reconstruct data from each source. Notationally, we represent joint entropy as $\mathcal{H}(X_1, X_2, \ldots, X_n)$, or simply $\mathcal{H}(\mathbf{N})$. Jointly coding data from correlated sources results in transmission of $\mathcal{H}(\mathbf{N})$ bits of information instead of $\sum_{i=1}^{n} \mathcal{H}(X_i)$. Note that $\mathcal{H}(\mathbf{N}) < \sum_{i=1}^{n} \mathcal{H}(X_i)$, in the existence of any data correlations. Temporal correlations further reduce data since only $\mathcal{H}(\mathbf{N}^t | \mathbf{N}^{t-1}, \ldots)$ (i.e., conditional entropy

[1] In a wireless network, a message broadcast transmission can be received by all nodes within the radio range of the broadcasting node.

of data at time t, given data from previous time steps) bits of information need to be transmitted, instead of $\mathcal{H}(\mathbf{N})$. In this section we focus primarily on spatial correlations. Temporal correlations can be computed from history buffers at source nodes.

2.2 Joint Entropy of Two Sources

The joint entropy of two source nodes is expressed as:

$$\mathcal{H}(X_1, X_2)=\mathcal{H}(X_1)+\mathcal{H}(X_2|X_1)=\mathcal{H}(X_1)+\mathcal{H}(X_2)-\mathcal{I}(X_1, X_2)\leq\mathcal{H}(X_1)+\mathcal{H}(X_2). \tag{1}$$

Here, $\mathcal{H}(X_2|X_1)$ is the conditional entropy of X_2 given X_1 and $\mathcal{I}(X_1, X_2)$ is the mutual information between the two random variables X_1 and X_2. Mutual information is a quantity that measures the correlation between two random variables.

As an example, consider a two-node system in which data transmitted by node 2 can be deterministically calculated using data from node 1. In this case, $\mathcal{H}(X_1, X_2) = \mathcal{H}(X_1)$, and only data from source 1 needs to be transmitted. Conversely, if no correlations exist (i.e., there is no mutual information), then $\mathcal{H}(X_1, X_2) = \mathcal{H}(X_1) + \mathcal{H}(X_2)$, and data from both sources must be transmitted. In sensor networks, data from one node is often correlated with nearby sources. Therefore, $\mathcal{H}(X_1, X_2) < \mathcal{H}(X_1) + \mathcal{H}(X_2)$. In such cases, only uncorrelated bits of X_2 (called error bits or ϵ) need to be transmitted to **exactly** reconstruct the data of node 2 using the data from node 1. In many cases, exact reconstruction of data is not required and a maximum error constraint, ϵ_m, can be provided by the user. In this case, if $\epsilon \leq \epsilon_m$, no data needs to be transmitted.

Mutual information in sensor networks quantifies correlations, which typically result from spatial locality of nodes in the network. Based on this spatial locality relation, mutual information for a pair of nodes can be expressed as $\mathcal{I}(X_1, X_2) = \mathcal{D}(d) \cdot min(\mathcal{H}(X_1), \mathcal{H}(X_2))$. Here, $\mathcal{D}(d)$, is a correlation scaling function defined in terms of the distance d between nodes 1 and 2, and takes values in the range $0 \leq \mathcal{D}(d) \leq 1$. The lower of the two source entropies provides a trivial bound on the maximum mutual information. The exact characteristics of the function $\mathcal{D}(d)$ depend on applications, and deployments within specific applications of sensornets. In typical applications, though, it is reasonable to expect that correlations are inversely related to proximity. We formally state this as:

Lemma 1. *Monotonicity of* $\mathcal{D}(d)$*:* $\mathcal{D}(d) \geq \mathcal{D}(d')$ *iff* $d \leq d'$.

We have, thus far, discussed the abstract correlation scaling function, $\mathcal{D}(\cdot)$ in terms of spatial distance, i.e., as $\mathcal{D}(d)$. However, in different deployments the scaling function may be defined in terms of other parameters, leading to the generalization $\mathcal{D}(\mathbf{R}_{i,j})$. Here, $\mathbf{R}_{i,j}$ represents the parameter set.

2.3 Joint Entropy of *N* Sources

A precise expression for (optimal) joint entropy must incorporate application features. To provide an application independent description, we define an approximation to the optimal joint entropy of the sensornet in terms of pair-wise mutual information. This approximation suffices to show that the spatial neighborhood model, which is the basis for SNP, achieves better compression than existing approaches. The joint entropy of the network is represented by $\mathcal{H}(\mathbf{N})$. Correlation-based compression induces an implicit ordering of sources. This is because data from a node i can only be coded with respect to the data from a node j, which itself is not encoded with respect to i. Therefore, an iterative construction[2] is required to evaluate H_n, our approximation to $\mathcal{H}(\mathbf{N})$:

Procedure 1. *Evaluating H_n.*

1. Initialize: Let, $\mathbf{S}$ be the set of nodes $\{v_2, v_3, \ldots, v_n\}$ and $\mathbf{V}$ be the set of a single node $\{v_1\}$. We set $H_1 = \mathcal{H}(X_{v_1})$, where X_{v_1} is the random variable associated with data from node v_1.
2. Iterate: for $i = 2 \ \mathit{to} \ n$
   - (a) Select node v_k from $\mathbf{S}$ and v_j from $\mathbf{V}$ such that $\mathcal{I}(X_{v_k}, X_{v_j})$ is maximized.
   - (b) Set $H_i = H_{i-1} + (\mathcal{H}(X_{v_k}) - \mathcal{I}(X_{v_k}, X_{v_j}))$.
   - (c) Remove node v_k from $\mathbf{S}$ and add it to set $\mathbf{V}$.
3. H_n is the approximation of $\mathcal{H}(\mathbf{N})$.

In the above procedure, step 2.a induces the ordering required for coding nodes with respect to each other. By selecting a node v_k that is maximally correlated (maximum mutual information) to some node in $\mathbf{V}$, this step minimizes joint entropy of the iteratively growing set, $\mathbf{V}$. H_n is an approximation to $\mathcal{H}(\mathbf{N})$ since pair-wise mutual information does not capture the information that v_k can extract from (all) other nodes in the set $\mathbf{V}$. Therefore, in general $\mathcal{H}(\mathbf{N}) \leq H_n$.

As stated earlier, the task of a distributed in-network compression protocol is to support efficient sharing of data from sources in the network to enable joint encoding of correlated data. In the context of Procedure 1, node v_k must have access to the data from node v_j. Furthermore, it is desirable that this data sharing be independent of the underlying network layout and routing topology. Existing approaches share data between nodes by partitioning the network into disjoint clusters and compressing the data at the cluster representative. However, this process has high compression overhead (i.e., data sharing overhead). Furthermore, the compression performance is sensitive to the optimality of partitioning.

2.4 Spatial Neighborhood Model (SN)

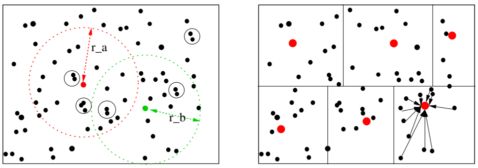


Fig. 1. (a) Overview of the SN model; (b) Overview of the PT model

The SN model is based on the following construction: let $\mathbf{S}$ be the set of n nodes in the network. For *each* node i, we define a spatial neighborhood set $\mathbf{S}_i^{r_i}$, which is a subset of $\mathbf{S}$ containing all nodes within distance r_i (except i itself). Here, r_i is called the correlation radius of node i. Corresponding to each set $\mathbf{S}_i^{r_i}$, we build a set of random variables $\mathbf{M}_i$, such that $\forall k \in \mathbf{S}_i^{r_i} : X_k \in \mathbf{M}_i$. From Lemma 1, nodes that are close to a given node i have a high spatial correlation

[2] This model is similar to the one presented in [8]. However, ours is a more general formulation.

with i. Therefore, the value of r_i can be chosen such the set $\mathbf{S}_i^{r_i}$ contains all nodes whose mutual information w.r.t. i is above threshold c. In the SN model, each node i receives messages from nodes in its spatial neighborhood set $\mathbf{S}_i^{r_i}$. Since a node k may be in several spatial neighborhood sets, it can communicate with all nodes i for which $k \in \mathbf{S}_i^{r_i}$ using a single message, assuming radio range exceeds r_i.

The construction of SN, thus far, implies that n broadcast messages are required, since each node must be in the spatial neighborhood set of at least one other node. However, due to redundancy in dense networks, we can prune the spatial neighborhood sets in such a way that a number of nodes (R) can be unaffiliated, i.e., are not in *any* set.

Definition 1. *A node $k \in \mathbf{S}_i^{r_i}$ is redundant w.r.t. to $\mathbf{S}_i^{r_i}$ if there exists a node $j \in \mathbf{S}_i^{r_i}$ such that $\mathcal{I}(X_i, X_k) \approx \mathcal{I}(X_i, X_j)$ and the mutual information between k and j is high (i.e., $\mathcal{I}(X_k, X_j) > c$, for some threshold c).*

To maximize R (and therefore, minimize message broadcast count), joint pruning of all the spatial neighborhood sets is needed. This is straightforward to achieve because, if the mutual information of two nodes, say k and j, is high, they are spatially close to each other. Therefore, the distance of node k and node j from another node i is approximately the same. Hence, $\mathcal{I}(X_i, X_k) \approx \mathcal{I}(X_i, X_j)$ for all other nodes i. Note that identifying R redundant nodes results in message reduction from n to $n - R$. This result is useful and important because redundancy typically increases with number of network nodes, implying that protocols based on the SN model scale well with increasing density.

Theorem 1. *Redundancy (R) increases monotonically with network density.*

Proof. Consider a pair of spatially proximate nodes k and j. From Lemma 1, their mutual information is potentially high, i.e., $\mathcal{I}(X_k, X_j) > c$. Furthermore, as these nodes come closer (increasing density), they belong to the neighborhood sets of an increasing number of nodes together. It then follows that $\mathcal{I}(X_i, X_k) \approx \mathcal{I}(X_i, X_j)$. Consequently, one of k or j can be removed from all spatial neighborhood sets (cf. Definition 1). It is easy to show that as the density of the network increases, the number of spatially proximate pairs increases linearly. One node from each such pair can be removed, if correlated, increasing R. If the network has uniform density this increase is linear as well. Note, though, that this relies on correlation (mutual information)–if there is no correlation, even with increasing density, R does not increase.

Figure 1(a) illustrates the SN model for a sample network layout. Two nodes a and b (shaded red and green, respectively) and their correlation radius r_a and r_b are shown. Solid (black outline) circles mark a few of the redundant pairs. One such pair is in the intersection of the correlation radius of nodes a and b. One of these nodes need not broadcast its data, without affecting the compression rate of the nodes a and b. We now show that the SN model achieves the bound set of joint entropy (i.e., compression rate) quantified by H_n.

Theorem 2. *The spatial neighborhood model (SN) achieves joint entropy, $\leq H_n$.*

Proof. This follows from the observation that the spatial neighborhood set of node i, $\mathbf{S}_i^{r_i}$, includes all nodes that have high mutual information w.r.t. node i (cf. Lemma 1).

Thus, it must include the node v_j from step 2.a of Procedure 1. Therefore, the SN model achieves joint entropy of $\leq H_n$.

An implication of the above theorem is that the SN model can achieve in-network compression such that at most H_n bits are transmitted to the sink. Let n_c denote the number of messages required to transmit H_n bits, and k be the average number of hops from the source to the sink (e.g., in a tree topology k is the height of the tree). Then the overhead of transmitting compressed data is $O(n_c \cdot k)$. As stated earlier, the overhead of data sharing in the network is $O(n - R)$. Therefore, we can derive the following theorem:

Theorem 3. *The network overhead of the SN model is $O(n - R) + O(n_c \cdot k)$.*

The key observation from this theorem is that the SN model scales well with increasing density of the network. This is because: (i) R increases with density, (ii) n_c decreases with density, and (iii) k remains approximately constant as the density increases.

2.5 Partitioning Model (PT)

An overview of the partitioning model for a random sensor network topology is presented in Figure 1(b). Here, the sensor network is portioned into m disjoint clusters of neighboring nodes in the network (in the figure $m = 6$). A cluster representative (shaded nodes in the figure), chosen within each partition, is responsible for receiving data from all nodes in the partition (see lower right partition). At each cluster representative, all collected data is then jointly coded and the compressed information is relayed to the sink. To evaluate the resulting joint entropy we let $\mathbf{C}_i$ be the set of random variables associated with the nodes in the i^{th} cluster. The joint entropy achieved by the PT model is therefore given by: $\mathcal{H}^{PT}(\mathbf{N}) = \sum_{i=1}^{m} \mathcal{H}(\mathbf{C}_i)$.

Theorem 4. *$\mathcal{H}^{PT}(\boldsymbol{N}) \geq H_n$.*

Proof. If the number of clusters $m = n$, then $\mathcal{H}^{PT}(\mathbf{N}) = \sum_{i=1}^{n}(\mathcal{H}(X_i)) \geq H_n$. If $m = 1$, clearly $\mathcal{H}^{PT} = H_n$. For any other value of m the entropy of each cluster can be found using Procedure 1; the sum of these entropies is greater than or equal to H_n.

Corollary 1. *The spatial neighborhood model (SN) achieves better compression rate than the portioning model (PT) because $\mathcal{H}^{SN}(\boldsymbol{N}) \leq H_n \leq \mathcal{H}^{PT}(\boldsymbol{N})$.*

Protocols based on the PT model (e.g., [2,5]) reduce the number of messages by pushing compression (or processing) of information into the network, i.e., to the cluster representatives. This decreases the network messages delivered to the sink. Unfortunately, such protocols still have a transmission overhead of $O(n \cdot k_h)$ because each node must necessarily transmit its data to the respective cluster representative. Here, k_h is the number of hops to the cluster representative. Thus, the compression overhead of the PT model is at least $O(n)$. Due to its construction, the PT model can not reduce this overhead to $O(n - R)$, which the SN model achieves. Therefore, in dense networks SN has significantly lower compression overhead than the PT model.

From Corollary 1, we see that the SN model achieves better compression than the PT model. Let, n_c be the number of messages required for transmitting the compressed data from the cluster representatives to the sink. Then, $O(n_c \cdot k_s)$ messages are required for transmitting data to the sink. Here, k_s is the average number of hops from the cluster representative to the sink. Recall that the SN model requires $O(n'_c \cdot k)$ messages, where $n'_c \leq n_c$ and in general $k > k_s$. However, as the density of the network increases, the $O(n)$ term dominates for PT, while $O(n - R)$ dominates for SN. Since R increases with density, the total overhead of SN is much lower. Therefore, the key aspect of the SN model is that it achieves a low compression overhead, while achieving similar compression rates as prior approaches.

3 The SNP Protocol

SNP is a distributed and self-organizing protocol that efficiently implements the SN model, achieving high associated compression rates at low overheads. It is practical and can be implemented on lean sensor nodes.

Protocol Overview. In the SN model, each node needs data from its neighbors, using which it can compress (correlate) its own data. This data is communicated through broadcasts. Other, correlated nodes, suppress their own broadcasts in response. The key unresolved issue is to construct a symmetric distributed coding and decoding scheme. Specifically, if node i codes its data w.r.t. node j, then node j may not code its data w.r.t. node i. Furthermore, the sink should be aware that node i used data from node j for reconstruction. Clearly, an ordering based on which coding can take place is required. The task of SNP is to induce such an ordering, while conforming to the SN model. The ordering induced by SNP is based on spatial relationships (and consequently, likelihood of correlation) between nodes.

Protocol Details. SNP partitions time into intervals of user-defined epochs (based on the data rate). Within each epoch, a node communicates (or suppresses its communication) at an allocated time. This time-ordering of nodes in a spatial neighborhood can be established using several protocols. SNP designates a subset of spatially distant (with distance $\approx D$ between them) nodes that initiate this ordering process. In this step, the designated nodes broadcast their data and go to sleep until the next epoch. Upon receiving this broadcast from a designated node, each node time-orders itself based on its distance from the designated node. This is done by initializing a count-down timer at node i to $T_i = \alpha \times d_{i,j}^2 + \beta_i$. Here, $\alpha \propto epoch/D^2$, $d_{i,j}$ represents the distance between node i and designated node j, and β_i is calculated using the hash of node id of i to the space $(0, \alpha)$. Note that SNP does not depend on the exact measurement of distance. A relative measure that is monotonic w.r.t. distance suffices. If nodes do not have a GPS, radio signal strength can be used [10]. Node locations can also be hard-coded into node IDs. The hash term, β prevents collisions between nodes that may be the same distance from a designated node. If a node receives messages from two nodes with different distances from it, the node chooses the closer of the two to synchronize its timer.

Once all timers have been initialized, we have an induced time-ordering of nodes in a spatial neighborhood (Figure 2). We refer to this ordering as Spatial Neighborhood Node ordering (SNO). This technique for deriving SNO has several desirable features: (i) it is resilient to node failures and insertions, (ii) it provides relative synchronization of the nodes and hence has much lower overhead than absolute time division and

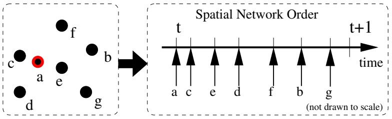


Fig. 2. Spatial neighborhood ordering in the SNP protocol. Node a initiates the ordering process.

synchronization protocols, (iii) it is independent of the radio range because nodes synchronize with messages from nearby neighbors. For the same reason it does not suffer from the hidden station problem, and (iv) it minimizes collisions in the network by providing a simple means of time division slotting (TDMA).

Prediction Functions. A prediction function, $\mathcal{F}_\theta$, estimates the data at a node from data at correlated sources: $\hat{x}_i^t = \mathcal{F}_\theta(x_i^t | x_j^t, x_k^t, \ldots, x_i^{t-1}, x_j^{t-1}, \ldots)$. Here, $\hat{x}_i^t$ is an estimate of x_i^t (the data at node i at time step t) computed from data at other nodes. Note that data from previous time steps (e.g., x_j^{t-1}) can also be used by the prediction function. The prediction error $|\epsilon|$ is given by $|x_i^t - \hat{x}_i^t|$. Higher correlation implies lower prediction error. Note that the prediction function has a model parameter θ_i for each node. These parameters are evaluated at the sink and transmitted to the nodes. Thus, the computationally intensive task of calculating parameters is performed at the sink, while the nodes use simple operations to predict data. θ_i can be updated at the sink if the correlations change. SNP is, itself, independent of the prediction function. The prediction function used in our implementation is discussed in Section 4.1.

Correlation Radius. Instead of defining correlation radius in terms of distance, SNP keeps two sets of nodes, $PRED_i$ (predecessors in time ordering) and $SUCC_i$ (successors in time ordering) at each node, that serve the same practical purpose. These sets are constructed locally at each node. For example, for node e in Figure 2, $PRED_e = \{a, c\}$ and $SUCC_e = \{g, b, f, d\}$. Note that these sets are sorted in terms of the distance of nodes from node e. A node can predict its data using data from the current time step from nodes in $PRED_i$ and data from the previous time steps using data from nodes $PRED_i \cup SUCC_i$. The number of nodes in these sets is γ_p (predecessors) and γ_s (successors), respectively. Large predecessor and successor sets improve compression, however, they also have associated memory overheads. In SNP, these parameters to be tunable by users. We show in our experiments that a small constant set size suffices in practice (Section 4).

Suppressing Data Broadcasts. A node determines whether it must broadcast its data or not based on the value predicted using its predecessors (and successors from prior epochs). This results in a self-adjusting mechanism, with varying density and correlations. *Due to this broadcast suppression mechanism, SNP achieves scalability with increasing density, as with the SN model.* The above mechanism is implemented using two thresholds, δ_l and δ_h. If the prediction error $|\epsilon| < \delta_l$, the node does not broadcast its data. In subsequent epochs, the node continues to suppress communication of its data unless $|\epsilon| > \delta_h$. This *hysteresis* based thresholding results in stability across slight correlation changes. Stability is an important part of this decision process, since a change in the decision at a node can affect the $PRED$ and $SUCC$ sets of other nodes. Conversely, if the decision process is over-damped, the system can not adapt to changing correlations. We show using experiments that this is not a major concern for SNP. Note that, as nodes broadcast their data, nodes that might not have heard the SNO initiators can set their timers based on messages heard from their neighbors and, thus, find their position in the SNO ordering. This overcomes the hidden station problem.

Data Compression and Transmission. Locally, each node i finds an estimate, $\hat{x}_i^t$, of its data x_i^t as: $\hat{x}_i^t = \mathcal{F}_\theta(x_i^t | \; PRED_i^t \; \cup SUCC_i^{t-1} \; \cup PRED_i^{t-1} \cup \ldots)$. The prediction error is given by $\epsilon_i^t = x_i^t - \hat{x}_i^t$. Users can specify ϵ_m, the maximum error tolerance (which can be zero). If $|\epsilon_i^t| \leq \epsilon_m$, no data is transmitted, otherwise only ϵ_i^t (which uses fewer bits) needs to be transmitted to the sink. Since data is communicated in packets, sending a packet with a few bits will have high overhead. Consequently, we buffer the prediction errors from multiple time epochs until the buffer is large enough to offset the packet overhead. We also use a threshold $thresh$, so that if $|\epsilon_i^t| > thresh$, the sensor measurement is immediately transfered to the sink[3]. In this manner, *outlier* data is immediately transmitted to the sink, while well correlated data is transfered lazily. *By using the data from both its predecessor and successors for compression, SNP faithfully implements the SN model and achieves compression rates of the SN model.*

The final step of the SNP protocol runs at the sink, which reconstructs data from compressed values, i.e., ϵ_i^t, received from each node. For this, the sink must be able to execute the same prediction operation. Once the estimate, $\hat{x}_i^t$, is evaluated at the sink, the actual value can be computed using the compressed bits received from the node. Clearly, for the sink to apply the prediction operation it needs to know the $SUCC_i$ and $PRED_i$ sets of a node. Recall that data at node x_i^t is predicted using data from the same time step from its $PRED_i$ set or data from previous time steps from its $PRED_i \cup SUCC_i$ set. Thus, all data required to re-construct x_i^t is available at sink. Each node communicates its $PRED_i$ and $SUCC_i$ sets to the sink. This needs to be done only once, when the sets are first constructed. This amortizes, over time, the overhead of communicating these lists. Note that these sets are stable because the broadcast suppression mechanism (which affects the nodes that can be in the $SUCC$ and $PRED$ sets) is stable, as discussed earlier and demonstrated in our experiments.

Resilience to Packet Losses. Packet losses can disrupt prediction, since data used for prediction at source may not have been received by the sink. Due to the use of spatial neighborhood ordering and data sharing, SNP minimizes packet losses from collisions and radio attenuation (due to spatial locality). Furthermore, the $PRED_i$ and $SUCC_i$ sets can be adapted so that nodes with repeated losses relative to the node i are removed from the sets.

4 Experimental Evaluation

We present a comprehensive evaluation of the performance of SNP over a 25 Mica2 node deployment, and using detailed simulations for parameter studies. We show that SNP provides up to 60% savings in network messages for fine-grained data collection.

[3] Data is transmitted to the sink using the underlying sensor network routing protocol, e.g., tree routing. Note that SNP is independent of this routing layer.

We compare SNP with existing approaches for in-network compression based on network partitioning, and show that these protocols require 25% to 50% more messages than SNP. Using simulation we evaluate the performance of SNP with increasing density and number of nodes in the network. Our results show that SNP scales well, exploiting both correlations and redundancy in dense networks. Finally, we evaluate the effect of different parameters of SNP on its performance and describe how they can be used to tune SNP for different environments.

4.1 Experimental Setup

We have implemented SNP on Mica2 nodes using COSMOS [1]. COSMOS supports a high-level programming model for sensor networks, with a lean runtime environment. The underlying source to sink data delivery uses tree routing. We present results using a lab testbed of 25 nodes. To evaluate SNP, we use data traces that are seeded on the sensor nodes. Thus, instead of sending data read from its sensors, the Mica2 nodes send data from the trace for repeatability. The trace, corresponds to temperature data from the Sonoma forest deployment [12].

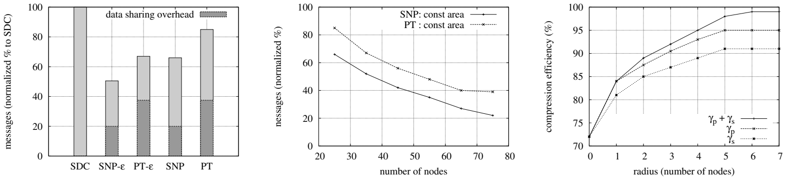
To enable a comparative study we also implement the PT protocol, and a simple data collection (SDC) protocol for baseline measurements. SDC does not use any in-network compression. The PT protocol implementation uses the same prediction function as the SNP protocol. To allow a fair comparison, we do not incorporate the cost of partitioning the network in the PT protocol. All other overheads, e.g., hop-by-hop messages due to the routing tree are incorporated. We also fix the tree routing structure so that measurements are comparable across runs. We have also built a simulator for the SNP, PT, and SDC protocols, which allows us to evaluate different operational ranges in detail.

Prediction Function. We use Autoregressive Moving Average (ARMA) based prediction to exploit spatio-temporal correlations. A node exploits data from multiple neighbors by taking a weighted average, or auto-regression, (based on spatial distance) of data. In addition to spatial correlations, each node exploits temporal correlations by maintaining a history of its own data and the data from its neighbors.

4.2 SNP Performance

We evaluate the performance of SNP in terms of compression overhead (messages) and rate w.r.t. the baseline SDC protocol and compare it to the PT protocol [2]. We also study the impact of approximate compression using the SNP-ϵ and PT-ϵ variants of the orignal protocols. In our experiments we use $\epsilon = 5\%$. We determine the number of messages for SNP, SNP-ϵ, PT, PT-ϵ and SDC protocols using our testbed of 25 Mica2 motes.

The results of this evaluation are shown in Figure 3(a). The number of messages are normalized to the number of messages required by SDC. As expected, in-network compression offers significant savings in the number of messages. SNP outperforms PT, reducing the message overhead by up to 30%. Furthermore, as expected, the approximate versions of the SNP and PT protocols perform better in terms of the message overhead. The key points to note is that the overhead of data sharing (shaded boxes) is significant. In fact, the superior performance of the PT protocol can be attributed mostly to the lower data sharing overhead. In all cases the overhead of data sharing in SNP is at



(a) (b) (c)

Fig. 3. (a) Number of messages (normalized w.r.t. SDC). (b) Messages in SNP and PT with increasing density. (c) Effect of changing predecessor and successor set size.

least 45% lower than the PT protocol. Another key point to note is that the data sharing overhead of SNP adjusts to correlations in the network, while that of the PT protocol remains the same. This is because, for the PT protocol, irrespective of the correlations in the network, each node must send its data to the cluster head for compression.

Impact of Node Density. We study the impact of node density on performance through simulations. To increase density the spatial area of the network (in the simulator) is kept constant while the number of nodes is increased. We observe that the ratio of messages required for SNP w.r.t. SDC tends to zero, while the same ratio for the PT protocol tends to a constant (Figure 3(b)).

Changing Correlation Radius. In the SNP protocol, each node maintains sets $PRED_i$ and $SUCC_i$, whose data is used to predict and, hence, compress data at the node. The sizes of these sets are γ_p, and γ_s, respectively. These parameters capture the correlation radius of a node and impact the memory-correlation tradeoff. We study the impact of γ_p and γ_s on compression. The results are presented in Figure 3(c). The metric of evaluation is compression efficiency, which is the ratio of the compression achieved using limited correlation radius with that of the compression achieved using an infinite radius. The three curves in the plots correspond to: (i) increasing γ_s while setting γ_p to zero, (ii) increasing γ_p with γ_s set to zero, and (iii) increasing $\gamma_p + \gamma_s$ with $\gamma_p = \lceil(\gamma_p + \gamma_s)/2\rceil$ and $\gamma_s = \lfloor(\gamma_p + \gamma_s)/2\rfloor$. We observe from Figure 3(c) the best performance is archived by using the $\gamma_p + \gamma_s$ approach, since using this approach, nodes are able to use the closest spatial neighbors. This is consistent with the intuition behind the construction of the SN model. An important implication of this result is that small sets are sufficient for achieving high (99%) efficiency.

5 Related Work

Application dependent in-network processing and aggregation based on data-centric routing has been well studied [6,7]. We develop an application independent in-network compression protocol to enable distributed joint coding, achieving high compression rate, with low overheads.

Traditional data compression schemes can not directly be adapted to sensor networks. There have been proposals to apply the much celebrated results of Selpian-Wolf [11] to sensor networks. Selpian-Wolf joint coding can achieve distributed compression without communication between the sources, which is attractive for sensor networks [9]. However, this approach requires precise *a priori* knowledge of the probability density function of data sources.

In early work on exploiting spatio-temporal correlations in sensed data, researchers have explored the PT model of correlation [2, 8]. We improve on these results, both in terms of compression rates and overhead. Pattem et al. [8] provide an information theoretic basis for the PT protocol. They also propose an implementation of the PT model in which compression can occur on the path to the cluster representative, thus, possibly reducing the $O(n)$ overhead of the PT protocol. However, this has the effect of increasing the route-length to the sink – thus adversely impacting network capacity. If this is not a consideration, the performance of this model approaches (but does not exceed) that of SNP.

6 Conclusions

In this paper, we present SNP, a novel application independent, lean, in-network compression protocol that achieves high compression rates by exploiting spatio-temporal correlations with low network overheads. We present formal quantification of compression rates, overheads, and scaling, and experimentally demonstrate its performance on real testbeds, as well as through simulations.

References

1. Awan, A., Jagannathan, S., Grama, A.: Macroprogramming heterogeneous sensor network systems using COSMOS. In: Proc. of EuroSys (March 2007)
2. Chu, D., Deshpande, A., Hellerstein, J.M., Hong, W.: Approximate data collection in sensor networks using probabilistic models. In: Proc. of ICDE 2006 (April 2006)
3. Levis, P., et al.: The Emergence of Networking Abstractions and Techniques in TinyOS. In: Proc. of NSDI 2004 (March 2004)
4. Gupta, P., Kumar, P.R.: The capacity of wireless networks. IEEE Transactions on Information Theory IT-46(2) (March 2000)
5. Heinzelman, W.R., Chandrakasan, A., Balakrishnan, H.: Energy-efficient communication protocols for wireless microsensor networks. In: Proc. of HICSS (January 2000)
6. Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., Silva, F.: Directed diffusion for wireless sensor networking. ACM/IEEE Transactions on Networking 11(1), 2–16 (2002)
7. Kulik, J., Rabiner, W., Balakrishnan, H.: Adaptive protocols for information dissemination in wireless sensor networks. In: Proc. of Mobicom 1999 (August 1999)
8. Pattem, S., Krishnamachari, B., Govindan, R.: The impact of spatial correlation on routing with compression in wireless sensor networks. In: Proc. of IPSN 2004 (April 2004)
9. Pradhan, S., Kusuma, J., Ramchandran, K.: Distributed compression in a dense microsensor network. IEEE Signal Processing Magazine 19(2) (March 2002)
10. Savvides, A., Han, C.-C., Strivastava, M.B.: Dynamic fine-grained localization in ad-hoc networks of sensors. In: Mobicom 2001 (July 2001)
11. Slepian, D., Wolf, J.: Noiseless coding of correlated information sources. IEEE Transactions on Information Theory 19(4)
12. Tolle, G.: Sonoma redwoods data (2005), www.cs.berkeley.edu/~get/sonoma

Task Scheduling on Heterogeneous Devices in Parallel Pervasive Systems (P^2S)

Sagar A. Tamhane and Mohan Kumar

Computer Science and Engineering Department,
University of Texas at Arlington, TX, USA
{sagar.tamhane,mkumar}@uta.edu

Abstract. Parallel Pervasive Systems (P^2S) comprise an ad hoc network of pervasive devices such as cell phones, handheld computers, laptops, sensors and other devices that essentially form a parallel system. Most of the current work in pervasive computing and mobile adhoc networks exploit resources on remote devices to execute compute intensive tasks. In this paper, we present a distributed task scheduling algorithm called cluster based scheduling algorithm (CBS) for parallel processing of task graphs in pervasive environments. We reduce the communication overhead by considering the devices to be grouped into logical clusters. CBS does not require the task scheduler device to have knowledge of all characteristics of each device in the environment. The proposed scheme allows usage of multiple task scheduling algorithms. Simulation results demonstrate time and energy efficient scheduling of tasks in heterogeneous environments.

Keywords: Task scheduling, parallel processing, pervasive computing, resource constrained devices.

1 Introduction

The concept of pervasive computing was prophesied by Mark Weiser in [1]. A pervasive environment provides services to its users in a transparent way such that the users get services whenever, wherever, however they want [2]. Each device in a pervasive environment possesses a set of hardware and software resources and serves as host to application services. Recent work [3] on creation, composition and maintenance of services in ubiquitous environments assume that the service provider device is capable of executing the service in its entirety. If the service provider is incapable of executing the service within the required quality of service (QoS), cyberforaging is used to transfer computations to a high-end server. We consider an environment in which such high-end servers are generally absent. Hence there is need for performing parallel processing using multiple devices that are available in the environment, rather than a single server. Parallel Pervasive Systems (P^2S) comprise sensors, monitors, cell phones, handheld computers, embedded devices, laptop and desktop computers and more. P^2S are

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 427–438, 2008.

essentially parallel and distributed computing systems that are networked using wired and/or various wireless technologies.

Parallel processing is a well-studied area for high-end machines. Parallel algorithms and programming techniques for several fundamental problems like matrix multiplication, sorting, searching, image analysis and several others have been described in the literature. Even though device parallelism exists in abundance in P^2S, exploiting parallelism using traditional algorithms is nontrivial due to such unique challenges as heterogeneity, resource constraints, mobility, and privacy and security. In particular, sensor and embedded systems are expected to be long running - meaning that once deployed they are expected to operate for long periods without human intervention. In pervasive systems, exploiting parallelism may increase the energy consumption due to increased communications. Therefore, it is necessary to develop novel techniques that can simultaneously meet time and energy constraints. To motivate the reader, consider the example of remote area surveillance using cameras and other sensors. Some of the sensors, such as a camera mounted on a patrol vehicle, would be mobile. Video and images streamed from the area need to be processed and only events of interests are transmitted. In such scenario, since the applications do not have access to high performance servers, it is necessary to exploit device parallelism in order to perform high computation tasks.

The proposed scheme, Cluster Based Scheduling (CBS), supports the device heterogeneity, mobility, and service fluctuations. Simulation results show that the communication overhead required by this scheme is far less than other schemes used in pervasive computing environments. CBS supports service fluctuations by considering each service to have a window during which it is available. Mobility is supported by considering each device to have a window during which the device is available.

1.1 Challenges in Parallel Computing in Pervasive Environments

User Mobility. Traditional parallel algorithms do not consider the participating processors to be mobile. In the above example, devices on the mobile patrol vehicle might be used for computation. These devices will move in/out of range as the vehicle moves. There will be constant changes in the underlying network of devices. In order to focus on task scheduling and resource utilization we assume that the pervasive environment can provide information about the expected arrival and departure times of a service and device.

Random Failures. Devices in pervasive environment can disconnect or fail at any time. There is no guarantee on the availability of those devices. Such random failures can be due to user turning off the device, device losing its network connectivity or device being nonresponsive due to some software error.

Device Heterogeneity. In a pervasive computing environment, the devices have varying CPU speeds and memory capacities. Dhodhi et al. [4] use a problem-space genetic algorithm for task scheduling on a heterogeneous set of processors. Only the CPU heterogeneity is considered. Other constraints like mobility and random failures are not considered.

Energy. Traditional parallel algorithms assume that there is sufficient energy to complete the assigned tasks. Mobile devices in a pervasive environment have limited energy remaining and might not be able to complete the tasks. Transmission and reception of a message consumes considerable amount of energy. Hence communication amongst processors should be reduced.

Interaction between User and Devices. Since personal devices are used for the parallel computations, the job should not inhibit the user from performing his tasks. For example, on a cellphone, an incoming phone call receives higher priority than the current parallel task. Hence it might not be possible to reserve the CPU for the entire amount of time of the parallel task.

Security. In a pervasive environment, when a client is using the neighboring devices for his parallelized tasks, the client should be protected from a malicious surrogate. Also, the surrogate needs to be protected from the job submitted by a client. If needed, the data should be encrypted to ensure privacy of the data.

1.2 Middleware Support

A typical middleware contains modules such as network handler, resource monitor, service discovery, service composition, application layer and security. The network handler provides the actual connections via various types of connectivity such as bluetooth, 802.11, LAN. A device can be connected to the rest of the network in various topologies such as ad-hoc, heirarchical, star, clusters. Algorithms such as [5] can be used for cluster formation and clusterhead election. The resource monitoring module measures resources like residual battery energy, residual storage space and others. The service discovery module locates services hosted by devices in the network. Mechanisms like [6] can be used in this module. The service composition module uses the services discovered to create new composite services by connecting the existing services in a task graph like fashion. Algorithms like [3] can be used in this module. The security module provides encryption and decryption algorithms.

2 Proposed Scheme

The proposed scheme requires devices to be grouped into logical clusters. Creation and maintenance of a cluster is provided by the middleware. Each cluster has a contact point or a cluster head that performs all housekeeping jobs for the cluster [3]. For the sake of simplicity, we assume a single cluster head per cluster. The cluster head can be distributed in space and/or time in order to address issues of load balance and fault tolerance. The creation of clusters ensures smooth execution of tasks in mobile and uncertain environments, where availability of a single node cannot be guaranteed for long periods of time. Fault tolerance and recovery will be investigated in the future.

Each device d_i is represented by the tuple $(\mu_i, tda_i, tdu_i, tdm_i, l_i)$ where μ_i: CPU speed of the device, tda_i: Time at which the device is available, tdu_i: Time at

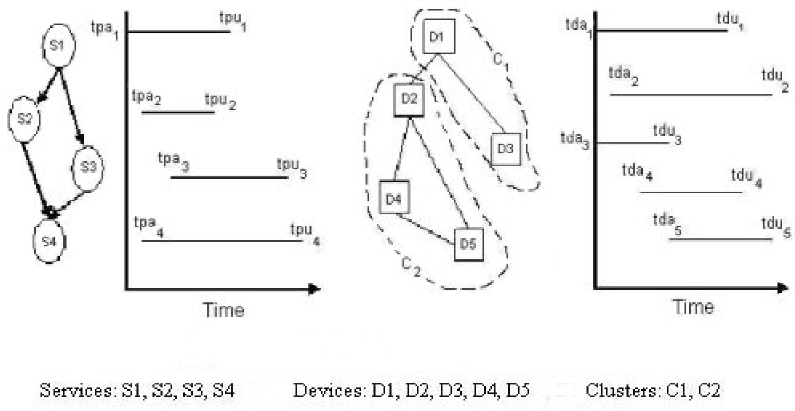


Fig. 1. Example task graph, device connectivity, availability of services and devices

which the device becomes unavailable, tdm_i: Time at which the device finished its previous task. Hence the device is available in the time interval $[tda_i, tdu_i]$, and $tda_i \leq tdm_i \leq tdu_i$. It is assumed that these values are known. l_i is the number of different voltage levels supported by the device for dynamic voltage scaling. Devices are grouped together into logical clusters based on some criteria. For example, the grouping would be on connectivity or services offered by the device or resource level of each device. Let C_x be a cluster of devices and represented as $\{d_1, d_2, ...\}$. Let $\mathbb{C} = \{C_1, C_2, ...\}$ be the set of all clusters.

Let $G = (P, E)$ be the directed task graph with $P = \{p_1, p_2, ...\}$ as the set of nodes or tasks and $e_{i,j} \epsilon E$ is an edge incident from p_i onto p_j. Each task p_j is represented by the tuple $(c_j, tpa_j, tpu_j, tps_j, tpf_j)$, where c_j: cpu requirement, tpa_j: time at which task p_j becomes available, tpu_j: time at which task p_j becomes unavailable, tps_j: time at which task p_j will start executing, tpf_j: time at which task p_j will stop executing. Let $r_{i,j}$ be the time required to pass data between the two tasks. When the two tasks are scheduled on same device, $r_{i,j} = 0$, otherwise $r_{i,j} \neq 0$. Let T be the deadline by which the entire task graph is to be executed. tpa_j and tpu_j are used to handle the service availability challenge. The pervasive environment might be able to provide these values. Figure 1 shows a sample task graph, device graph and availability windows of each service and device. The scheduling is performed in two levels: client - cluster communication and intra-cluster scheduling.

2.1 Client - Cluster Communication

A device that initiates the task scheduling is called as the *client*. Algorithm 1 gives the communication protocol between the client and the cluster heads. The client divides the deadline, T, into $|\mathbb{C}|$ equal intermediate deadlines. Let $\mathbb{T}$ be the set of these deadlines. Hence, $\mathbb{T} = \{T/|\mathbb{C}|, 2T/|\mathbb{C}|, 3T/|\mathbb{C}|, ..., T\}$. The client informs all the clusters of the task graph and the intermediate deadlines. Each

Algorithm 1. Client-Cluster communication

```
$\mathbb{T} = \{T/|\mathbb{C}|, 2T/|\mathbb{C}|, 3T/|\mathbb{C}|, ..., T\}$
for all $t$ in $\mathbb{T}$ do
    send (winner of prev. slot, request for bids for current slot) to all cluster heads.
    receive (bids from cluster heads)
end for
```

cluster head performs task scheduling using devices in that cluster only. Depending on the resources available within the cluster, the clusterhead uses a suitable task scheduling algorithm. Greedy and list based algorithms require less CPU and energy to generate a schedule whereas genetic algorithms can generate better quality schedules in more time.

Each cluster returns the number of tasks that can be completed by the intermediate deadline and the start time and finish time of the tasks selected in a particular time slot. This is called as a *bid*. The cluster that performs the maximum number of tasks by the intermediate deadline is declared as the winner of that time slot. If two clusters select same number of tasks, then the cluster that will execute the tasks in the least time is selected. If multiple clusters finish the same number of task at the same time, the tie is broken randomly. The result along with the start and finish time of the selected tasks is then sent to all clusters and the bidding process for the next time slot begins. Hence the number of messages received (rx) and transmitted (tx) by the client are: $rx = |\mathbb{C}|^2$, $tx = rx + 1$ respectively.

2.2 Intra-cluster Scheduling

Due to the user mobility patterns and energy remaining on each device, each device will have a time window during which it is available for parallel computing. We divide the intermediate deadline provided by the client into more intervals depending on the arrival and departure time of the devices. The tasks are sorted on their bottom level (b-level) [7] to get the preferred sequence in which the tasks should be scheduled. Algorithm 2 describes an intra-cluster scheduling algorithm. Let τ_a be the set of device available times, and τ_u be the set of device unavailable times. Hence t is the sorted sequence of events of device becoming available or unavailable within a particular cluster, where t_v is the v^{th} element of t and $v < w \Leftrightarrow t_v \leq t_w$. Hence each device is either available during the entire interval or is unavailable during the entire interval. $\mathbb{D}$ contains devices that can be used till time t_v. Step 6 calculates ϱ, the total computing power available till time t_v. Since each iteration performs scheduling in only one interval, we can reduce the task graph such that tasks that require more computing power than ϱ can be ignored. Let $\mathbb{P}$ be the set of unscheduled tasks such that sum of CPU requirements of tasks in $\mathbb{P}$ is less than or equal to $\varrho.\lambda$, where λ is the selection factor. Thus $\mathbb{P}$ contains tasks that could be scheduled till time t_v. Since the tasks are sorted according to their distance from the exit node, λ controls the

Algorithm 2. Intracluster scheduling

```
1. $\tau_a = \{tda_1, tda_2, ...\}$ and $\tau_u = \{tdu_1, tdu_2, ...\}$
2. $t = \{\tau_a \cup \tau_u\}$
3. for all $t_v \leq \mathbb{T}_h$ do
4.    $\mathbb{D} \subseteq C_x$ such that $d_i \in \mathbb{D} \Leftrightarrow tdm_i < t_v$
5.    sort $\mathbb{D}$ in descending order of $\mu_i$
6.    $\varrho = \Sigma_{d_i \in \mathbb{D}}[\mu_i(t_v - tdm_i)]$
7.    $\mathbb{P} \subseteq P$ such that $\Sigma_{p_j \in \mathbb{P}} c_j \leq \varrho.\lambda$
8.    $\rho^j \subset \mathbb{P}$ such that $\rho^j_k \in \rho^j \Rightarrow \rho^j_k$ is an unscheduled parent of $p_j$
9.    for all $d_i \in \mathbb{D}$ do
10.      $\psi \subseteq \mathbb{P}$ where $\psi_j \in \psi \Rightarrow (\Sigma_{p_k \in \rho^j} c_k) + c_j \leq [\min(tpu_j, t_v) - \max(tpa_j, tdm_i)].\mu_i$
11.      Consider the device to be a knapsack of capacity $\mathbb{r}$. Form a table of $\mathbb{c} = |\psi|$
         columns and $\mathbb{r}$ rows
12.      Solve knapsack using recurrence relation: $[\mathbb{r}, \mathbb{c}] = \max\{[\mathbb{r}, \mathbb{c} - 1], ([\mathbb{r} - \psi_j, \mathbb{c} -$
         $1] + \psi_j$ if $\rho^j \in [\mathbb{r} - \psi_j, \mathbb{c} - 1] \bigwedge \max_{p_k \in \rho^j}(tpf_k + r_{k,j}) + c_j/\mu_i \leq \min(tpu_j, t_v))\}$
13.      $\phi_{r,c} = (\Sigma_{e_{k,j} \in Es_{r,c}}(r_{k,j}) + \Sigma_{p_j \in [\mathbb{r},\mathbb{c}]}(c_j))/\Sigma_{e_{k,j} \in Ec_{r,c}}(r_{k,j})$
14.      Choose $[\mathbb{r}, \mathbb{c}]$ with highest rank. Schedule corresponding tasks onto $d_i$
15.      Remove those tasks from $P, \mathbb{P}$
16.      Update $tps_j, tpf_j$ of the tasks and $tdm_i$ of the device
17.      Send the $tps_j, tpf_j$ values to the client
18.   end for
19. end for
```

number of edges that have both ends in $\mathbb{P}$. If the value of λ is small, we might get only disconnected tasks in $\mathbb{P}$. Scheduling both ends of the edges onto same device reduces the communication cost. Figure 2 shows an example of the effect of values of λ. Let ρ^j be the set of unscheduled parents of p_j. Steps 10 to 18 are repeated for each device available till time t_v. A task might be scheduled onto d_i if the sum of the CPU requirement of the task and its unscheduled parents is less than or equal to the computation power available with the device in the interval $[tdm_i, t_v]$. Let ψ be the set of such tasks and ψ_j be the j^{th} element in ψ. Hence $\psi_j \in \psi \Rightarrow (\Sigma_{p_k \in \rho^j} c_k) + c_j \leq [\min(tpu_j, t_v) - \max(tpa_j, tdm_i)].\mu_i$. To select tasks for d_i, the computation power offered by the device till t_v is considered as the

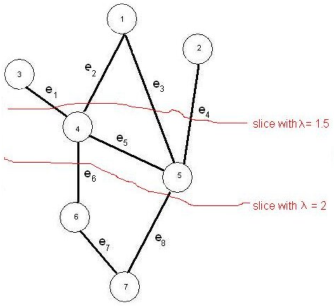


Fig. 2. Example: Selection of tasks into $\mathbb{P}$ based on value of λ

capacity of the knapsack. Hence capacity of the knapsack is $(t_v - tdm_i)\mu_i$. A dynamic programming table for the knapsack problem gives better results when the rows of the table are numbered consecutively, starting from 0. We create a dynamic programming table of $\mathbb{c} = |\psi|$ columns and $\mathbb{r}$ rows. For simulation purpose we considered $\mathbb{r} = 10$. Since the capacity of knapsack is $(t_v - tdm_i)\mu_i$, each row in the table will correspond to a multiple of $((t_v - tdm_i)\mu_i)/\mathbb{r}$. Since the knapsack capacity has been converted into $\mathbb{r}$, the CPU requirement of each task in ψ should be converted in the corresponding ratio. Hence divide $c_j \in \psi$ by $((t_v - tdm_i)\mu_i)/\mathbb{r}$. Step 12 gives the recurrence relation for the dynamic programming table. Let $Ec_{r,c}$ be the cutset of the combination $[\mathbb{r}, \mathbb{c}]$. Let $Es_{r,c}$ be the edges that have both ends in $[\mathbb{r}, \mathbb{c}]$. The rank of combination of tasks in $[\mathbb{r}, \mathbb{c}]$ is given by

$$\phi_{r,c} = \frac{\Sigma_{e_{k,j} \in Es_{r,c}}(r_{k,j}) + \Sigma_{p_j \in [\mathbb{r},\mathbb{c}]}(c_j)}{\Sigma_{e_{k,j} \in Ec_{r,c}}(r_{k,j})} \quad . \tag{1}$$

The combination with highest rank is scheduled onto device d_i. The tps_j, tpf_j and the number of tasks in the selected combination are sent to the client.

Consider the example task graph of Figure 2. Let $\lambda = 2$, $\varrho = 8$, $c_j = j$ and $e_j = j$, that is, $c_1 = 1$, $c_2 = 2$, $c_3 = 3, \ldots,$ and $e_1 = 1$, $e_2 = 2$, $e_3 = 3, \ldots$. Hence rank of combination $(1, 2, 3) = 0.6$ and rank of combination $(1, 3, 4) = 0.785$. Thus varying the value of λ might give us better combinations. Consider that a schedule is obtained using the above method. For each device, the sequence of tasks assigned to the device is checked for any idle time between two tasks. If an idle time is found and the corresponding device supports dynamic voltage scaling (DVS), the previous task may be executed on a lower voltage such that the idle time is minimised. A task is not considered for DVS if its output is sent to another task on a different device. Consider an idle time between tasks p_i and p_j. Let the device support k different voltage levels. The following steps are repeated till the idle time is minimised:

1. Let τ_k be the total time required to execute p_i at voltage level, k.
2. If $\tau_k \leq (tps_j - tps_i)$, decrement k by 1. Else execute p_i using voltage level $k + 1$.

Let E_j be the energy savings obtained by executing task p_j on a lower voltage level. The total energy savings for a schedule is $\sum_j E_j$.

In Algorithm 1, consider that T is not divided into slots. Hence bidding will be done only once and the obtained schedule consists of all tasks mapped onto devices in a single cluster. Depending on the device characteristics, a better solution might be obtained if the tasks are scheduled across multiple clusters. Hence instead of having just one slot, we divide T into multiple slots. Since bidding is performed for every slot, the communication overhead increases with increasing number of slots. Hence we use $|\mathbb{C}|$ number of slots as a tradeoff. Algorithm 2 is executed once for each slot of Algorithm 1. Hence Algorithm 2 is executed $|\mathbb{C}|$ times. Let there be n devices and m tasks. Algorithm 2 sorts the arrival and departure time of devices in ascending. Devices that are available in each interval are found and the innermost loop is repeated for each of that device. We get maximum number of intervals when only 1 device arrives or departs at an instance. Hence if there are x devices in a cluster, the innermost loop is executed x^2 times for the worst case. The inner loop creates a dynamic programming table of size y rows $*|\psi|$ columns. The worst case will be when $\psi = \mathbb{P}$, that is $|\psi| = m$. Hence it will take $2ym$ computations for each iteration of the inner loop. After a schedule is obtained from algorithm 2, each task might be executed on a lower voltage level. Hence it takes $\sum_j l_{\delta(j)}$ computations, where $\delta(j)$ is the device on which p_j is scheduled. Let $l_{max} = \max_i(l_i)$. Hence worst case complexity of the algorithm is $O(y.m.|\mathbb{C}|.x^2 + m.l_{max})$. The client transmits and receives a total number of $2|\mathbb{C}|^2 + 1$ messages. Thus we see that if keeping n as a constant, if we increase $|\mathbb{C}|$, the communication cost increases and the computation cost decreases. The total number of messages sent by all devices in a work stealing algorithm like [8] is given by $(n^2).f$, where f is the frequency of advertisement messages sent. Thus we see that even though the computation complexity is higher in our algorithm, the energy savings obtained due to reduced communication will be much higher and the higher computation complexity is worth the energy savings.

3 Simulation Studies

We compare CBS with the HEFT algorithm [9] used over all devices. To evaluate the communication overhead and energy consumption, we compare CBS with algorithms given in [10] and [8]. The simulations were performed for random task graphs and randomly generated device characteristics and connectivity.

3.1 Comparison Metrics

- Normalized Schedule Length (NSL): NSL is defined as the makespan divided by the minimum time required to execute the critical path. $NSL \geq 1$ since the denominator is the lower bound of the makespan. We use average NSL over 25 sets of task and device graphs.

$$NSL = \frac{makespan}{\min_{i=1,\ldots,n}\left(tda_i + \frac{\sum_{p_j \epsilon cp} c_j}{\mu_i}\right)} \ . \tag{2}$$

  where *cp* is the critical path of task graph.
- Efficiency: Speedup is defined as the ratio of sequential execution time of the entire task graph on a single device to the parallel execution time. Efficiency is defined as speedup divided by the number of devices used in the schedule.

$$speedup = \frac{\min_{i=1,\ldots,n}\left(tda_i + \frac{\sum_{p_j \epsilon P} c_j}{\mu_i}\right)}{makespan} \ . \tag{3}$$

– Energy consumed in communication overhead: We find the ratio of energy consumed by algorithm in [10] to that consumed by CBS and the ratio of energy consumed by MPI-IOS [8] to that consumed by CBS.

3.2 Random Graph Generation

Random graphs are generated with the following parameters:

- Number of tasks (n): The number of tasks is selected from the set $\{20, 40, 60, 80, 100\}$.
- Computation cost (c): Each task in the task graph has CPU requirement in the range of 1 to 200 units.
- Communication to computation ratio (CCR): The average value of CCR is selected from the set $\{0.01, 0.1, 1, 10, 100\}$. Communication is lightweight if the ratio is 0.01 and is most heavy when the ratio is 100.
- Number of devices (m): The number of devices considered are 4, 8, 16, 32 or 64.
- CPU speed of each device (μ): The CPU speed is chosen randomly from the range 1 to 600 units.

Figure 3 shows the effect of CCR on average NSL for $CCR = 0.1, 1, 10$. We see that irrespective of the CCR, our scheme performs better than HEFT and gives a lower average NSL. For many samples, HEFT did not generate a schedule, but our scheme did. Such cases were not considered for the average NSL in figure 3. In these samples, there was a task with multiple parents scheduled onto multiple devices. The communication cost between devices caused finish time of the child task to become greater than availability of all devices. Our scheme uses a knapsack dynamic programming table and is not based on the earliest finish time. Hence less number of devices were used for the parents and hence the finish time of the child task was within the device availibility window.

Figure 4 shows the efficiency comparison between HEFT and our scheme. We see that our scheme consistently produces higher efficiency. The number of devices used ranged from 4 to 64. We compared the communication overhead cost in CBS with that in [10] and MPI-IOS. In [10], a device that executes a particular task becomes responsible for scheduling the successors of that task. This requires communication between a task and all other devices. Due to clustering, we were

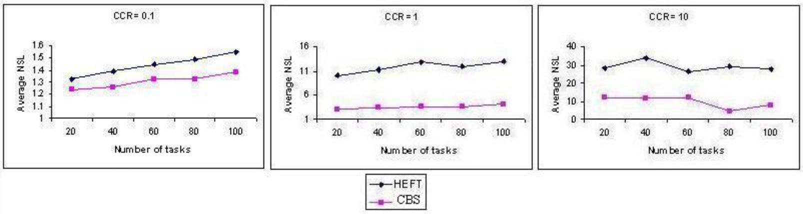


Fig. 3. Effect of CCR on Average Normalized Schedule Length

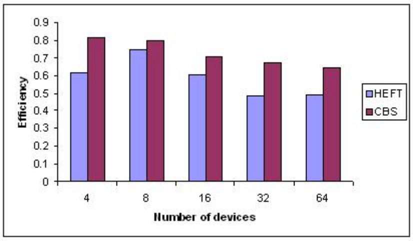


Fig. 4. Efficiency comparison ($CCR = 0.01, n = 20$)

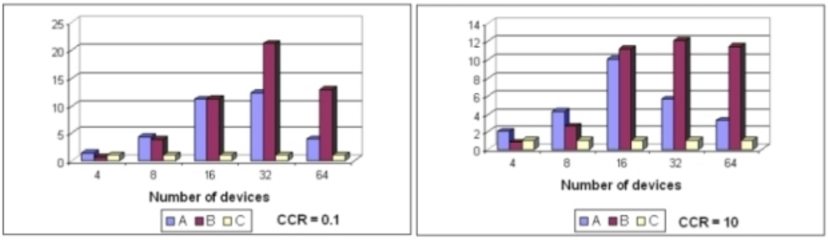


Fig. 5. Ratios of energy spent in communication overhead of A: algorithm in [10], B: MPI-IOS algorithm with respect to C: CBS

able to reduce the communication required. For figure 5, we find the ratio of energy consumption in MPI-IOS and the algorithm in [10] to that of CBS. Figure 5 shows the ratios for $CCR = 0.1$ and $CCR = 10$ with $n = 20$ and m ranging from 4 to 64. Number of messages exchanged in [10] and CBS are dependant on device connectivity and number of clusters respectively. Number of messages exchanged in MPI-IOS depends not only on device characteristics but also on the time taken to finish the task graph. Hence we observe that energy consumed by MPI-IOS is much larger than that in [10] and CBS.

4 Related Work

Phan et al. [11] consider mobile devices for grid computing. However, all of the challenges presented by a pervasive computing environment are not handled. Yu et al [12] propose an energy-balanced allocation algorithm for homogeneous sensor nodes equipped with dynamic voltage scaling. All devices are assumed to be connected with each other and available till the execution of all tasks is over. Cyber foraging [13] is a method for migrating a task from a mobile device to a resourceful server. To make an application usable for cyber foraging, the application developer modifies the application such that it is divided into parts that can be executed on a remote server. DynaMP [14] is a message passing architecture used for parallel computation in mobile systems. DynaMP uses all its neighbors in a Bluetooth piconet. Work is distributed evenly amongst all the neighbors.

MagnetOS [15] partitions applications into components and dynamically allocates them to nodes within the network. The granularity of partitioning is a Java object. In [8], the authors demonstrate how to achieve process migration in applications that use message passing interface (MPI). When a node becomes underutilized, it generates work stealing requests. Each such request has the performance-related information about that node. Upon receiving such request, each node decides whether there are any processes that can be migrated. In [5], cluster formation and clusterhead election and failure recovery for adhoc networks has been investigated. In [16], the authors perform task scheduling over a heterogeneous collection of homogenous clusters. Devices in a cluster are similar in their characteristics. Devices in different clusters might be different.

5 Conclusion and Future Work

In this paper we have proposed a novel cluster based algorithm for task scheduling in parallel pervasive systems. We have shown that the algorithm has less communication overhead for task scheduling. Using simulation results, we have compared our algorithm with two other schemes. The proposed cluster based scheme overcomes some of the challenges - device heterogeneity, service availability and device mobility common to mobile ad hoc networks and pervasive environments. The algorithm allows the use of multiple scheduling heuristics depending on the capabilities of the devices and cluster head.

In future, we envisage to add fault tolerance features to the algorithm. An algorithm for handling the user-device interaction challenge will be investigated. Above simulations used clustering based on device connectivity. We are analysing algorithms that perform clustering based on device functionality and the offered services.

References

1. Weiser, M.: The computer for 21st century. J. Scientific American. 265(3), 94–104 (1991)
2. Ark, W.S., Selker, T.: A look at human interaction with pervasive computers. IBM Systems Journal 38(4), 504–508 (1999)
3. Kalasapur, S., Shirazi, B.A., Kumar, M.: Dynamic Service Composition in Pervasive Computing Systems. IEEE Transactions on Parallel and Distributed Systems 18(7), 907–918 (2007)
4. Dhodhi, M.K., Ahmad, I., Muhammad, A.Y., Ahmad, I.: An Integrated Technique for Task Matching and Scheduling onto Distributed Heterogeneous Computing Systems. J. Parallel and Distributed Computing. 62(9), 1338–1361 (2002)
5. Chatterjee, M., Das, S.K., Turgut, D.: WCA: A Weighted Clustering Algorithm for Mobile Ad Hoc Networks. J. Cluster Computing. 5(2), 193–204 (2002)
6. Waldo, J.: The Jini architecture for network-centric computing. Communications of the ACM 42(7), 76–82 (1999)
7. Kwok, Y.-K., Ahmad, I.: Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. ACM Computing Surveys 31(4), 406–471 (1999)

8. Maghraoui, K.E., Desell, T., Szymanski, B.K., Teresco, J.D., Varela, C.: Towards a Middleware Framework for Dynamically Reconfigurable Scientific Computing. In: Grandinetti, L. (ed.) Grid Computing and New Frontiers of High Performance Processing. Advances in Parallel Computing, vol. 14, pp. 275–301. Elsevier, Amsterdam (2005)
9. Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems 13(3), 260–274 (2002)
10. Bauch, A., Maehle, E., Markus, F.-J.: A Distributed Algorithm For Fault-tolerant Dynamic Task Scheduling. In: Second Euromicro Workshop on Parallel and Distributed Processing, pp. 309–316 (1994)
11. Phan, T., Huang, L., Dulan, C.: Challenge: Integrating Mobile Wireless Devices Into the Computational Grid. In: 8th ACM International Conference on Mobile Computing and Networking (Mobicom), pp. 271–278 (2002)
12. Yu, Y., Prasanna, V.K.: Energy-Balanced Task Allocation for Collaborative Processing in Wireless Sensor Networks. ACM/Kluwer J. Mobile Networks and Applications (MONET) Special Issue on Algorithmic Solutions for Wireless, Mobile, Ad Hoc and Sensor Networks 10(1), 115–131 (2005)
13. Balan, R.K., Gergle, D., Satyanarayanan, M., Herbsleb, J.: Simplifying Cyber Foraging for Mobile Devices. Technical Report CMU-CS-05-157R. Carnegie Mellon University (2005)
14. Shepherd, R., Story, J., Mansoor, S.: Parallel Computation in Mobile Systems using Bluetooth Scatternets and Java. In: International Conference on Parallel and Distributed Computing and Networks (2004)
15. Liu, H., Roeder, T., Walsh, K., Barr, R., Sirer, E.G.: Design and implementation of a single system image operating system for ad hoc networks. In: 3rd International Conference on Mobile Systems, Applications, and Services, pp. 149–162 (2005)
16. N'Takpe, T., Suter, F.: Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms. In: 12th International Conference on Parallel and Distributed Systems (ICPADS), pp. 3–10 (2006)

A Performance Guaranteed Distributed Multicast Algorithm for Long-Lived Directional Communications in WANETs

Song Guo[1], Minyi Guo[1], and Victor Leung[2]

[1] School of Computer Science and Engineering, University of Aizu
Aizu-Wakamatsu, Fukushima, 965-8580, Japan
{sguo,minyi}@u-aizu.ac.jp
[2] Department of Electrical and Computer Engineering, University of British Columbia
Vancouver, British Columbia, Canada V6T 1Z4
vleung@ece.ubc.ca

Abstract. We consider the lifetime optimization problem for multicasting in wireless ad hoc networks, in which each node is equipped with a directional antenna and has limited energy supplies. In this paper, we propose a new distributed algorithm, whose performance in terms of providing long-lived multicast tree is guaranteed by our theoretical analysis. We prove that its approximation ratio is bounded by a finite number. In particular, the derived upper bound in a closed form shows that the algorithm can achieve global optimal in some cases. The real performance of this new proposed algorithm is also evaluated using simulation studies and the experimental results show that it outperforms other distributed algorithms.

Keywords: Wireless Ad Hoc Network, Multicast, Directional Antenna, Energy Efficiency, Approximation Algorithm.

1 Introduction

Energy conservation is of paramount importance for the wide deployment of wireless ad hoc networks (WANETs) in the forms of mobile ad hoc networks (MANETs) and wireless sensor networks (WSNs) due to their potentially extensive civil and military applications. Multicasting plays an important role in typical WANETs where bandwidth is scarce and hosts have limited battery power. In addition, many routing protocols for MANETs need a broadcast / multicast as a communication primitive to update their states and maintain the routes between nodes. Multicast is also widely used in WSNs to disseminate information, *e.g.* environmental changes, to other nodes in the network. Therefore, it is essential to develop efficient multicast protocols that are optimized for energy consumption. There are two energy-aware metrics and their corresponding problem formulations that have been most widely studied: (1) to minimize the energy consumption and (2) to maximize the network operating lifetime.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 439–450, 2008.

Both problems have received equal attentions, *e.g.* the work [1-6] for the first problem and [7-15] for the second. In this paper we have focused on the second problem.

The network operating lifetime is typically defined as the duration of the network operation time until the battery depletion of the first node in the network. Some work has considered maximizing the network lifetime in a network for broadcast session, *e.g.* [7-10], or for multicast session, *e.g.* [10-14]. Some optimal solutions [12-14] with polynomial time complexity show that such optimization problem belongs to P. Over the last few years, energy efficient communication in wireless ad hoc networks with directional antennas has received more and more attention. This is because directional communications can save transmission power by concentrating RF energy where it is needed [17, 18]. The same optimization problem using directional antennas has been studied in [15-20] and has been proven to be a NP-hard problem [20]. The exact solution for such difficult problem is presented in [19] based a MILP (mixed integer linear programming) formulation.

The most desirable work [16] proposed two distributed algorithms DMMT-OA/DMMT-DA (Distributed Min-Max Tree algorithm for Omnidirectional / Directional Antennas) to provide long-lived multicasting in WANETs with directional antennas. Simulation results have also shown that these two distributed multicast algorithms for directional communications outperform other centralized multicast algorithms, *e.g.* in [15, 17, 18]. The advance of this work inspires us to further investigate the distributed solutions for this optimization problem. A careful observation on the DMMT-DA algorithm leads to a new distributed algorithm with improved performance. The proposed algorithm uses a node-centric point of view, in stead of the traditional link-centric manner [15, 16], such that it can avoid some cases that are far from optimal. We then use a graph theoretic approach to analysis its theoretical performance in terms of approximation ratio. The derived bound, in a closed analytical expression, of this approximation ratio shows that our proposed algorithm is a constant-factor approximation algorithm. In order to evaluate the real performance of our proposed algorithm, we use simulation as well to compare against a set of distributed algorithms and find that it outperforms other proposals.

2 System Model and Problem Formulation

We model our wireless ad hoc network as a simple directed graph G with a finite node set N ($|N| = n$) and an arc set A corresponding to the unidirectional wireless communication links. Each node is equipped with a directional antenna, which concentrates RF transmission power to where it is needed. We assume a widely used propagation model [1] for adaptive antennas [15-18], in which the antenna at each node v can switch its orientation to any desired direction with transmission power uniformly distributed across its adjustable beamwidth θ_v between $\theta_{\min}$ and 2π. The transmission power p_{vu} to support a link (v, u) separated by a distance r_{vu} ($r_{vu} > 1$) is therefore proportional to r_{vu}^{α} and θ_v with unit signal detection threshold, where the propagation loss exponent α typically takes on a value between 2 and 4. We further assume that any node $v \in N$ can choose its transmission power, strictly within some minimum and maximum levels $p_{\min}$ and $p_{\max}$, respectively, which are positive constant numbers. The transmission power p_{vu} thus can be expressed as follows.

$$p_{vu} = p(r_{vu}, \theta_v) \tag{1}$$

$$p(r,\theta) \equiv \max(p_{\min}, \theta \cdot r^{\alpha} / 2\pi) \leq p_{\max} \tag{2}$$

Let $\varepsilon = \{\varepsilon_v > 0 \mid v \in N\}$ be the energy supply associated with each node in G. The residual lifetime τ_{vu} of an arc $(v, u) \in A(T_s)$ is therefore ε_v / p_{vu}.

We consider a source-initiated multicast with multicast members $M = \{s\} \cup D$ ($|M| = m$), where s is the source node and D are destination nodes. All the nodes involved in the multicast form a multicast tree rooted at the node s, *i.e.* a rooted tree T_s, with a tree node set $N(T_s)$ and a tree arc set $A(T_s)$. We define a rooted tree as a directed acyclic graph with a source node with no incoming arcs, and each other node v has exactly one incoming arc. A node with no out-going arcs is called a leaf node, and all other nodes are internal nodes (also called relay nodes). We use $\Lambda_v^+(T_s)$ and $\lambda_v^+(T_s)$ to denote the child node set and the out-degree (*i.e.* the number of child nodes) of node v in the tree T_s, respectively.

Let Ω_M be the family of all rooted multicast trees spanning nodes in M. The maximum-lifetime multicast problem can thus be expressed as

$$\max_{T_s \in \Omega_M} \min_{(v,u) \in T_s} (\tau_{vu}) = 1 / \min_{T_s \in \Omega_M} \max_{(v,u) \in T_s} (1/\tau_{vu}) \,. \tag{3}$$

Note that if we assign the tree arc weight function w_{vu} as the reciprocal of the lifetime of the arc (v, u), *i.e.*

$$w_{vu} = 1/\tau_{vu} = p(r_{vu}, \theta_v) / \varepsilon_v \,, \tag{4}$$

our optimization problem is equivalent to the *min-max tree problem*, which is to determine a directed tree T_s including all the multicast members (*i.e.*, $M \in N(T_s)$) such that the maximum arc weight is minimized. The corresponding optimal solution is just the reciprocal of the lifetime of the maximum-lifetime multicast tree.

Given a multicast tree T_s, we use $\delta_o(T_s)$ and $\delta_d(T_s)$ to denote the maximum arc weight of the same tree in a network instance $G(N, A)$ with omni-directional antennas and directional antennas, respectively, *i.e.*

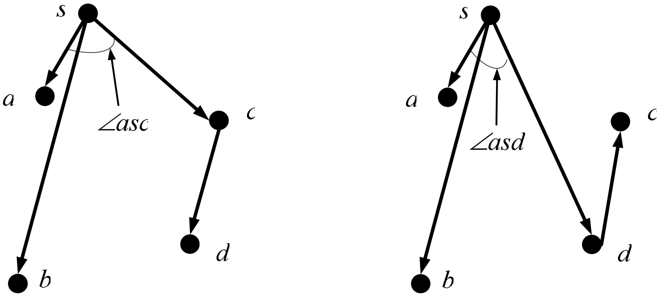
$$\delta_o(T_s) \equiv \max_{(v,u) \in A(T_s)} (p(r_{vu}, 2\pi) / \varepsilon_v) \,, \tag{5}$$

$$\delta_d(T_s) \equiv \max_{(v,u) \in A(T_s)} (p(r_{vu}, \theta_v) / \varepsilon_v) \,. \tag{6}$$

The arc with the above weights (5) and (6) is called the *omni-directional* and *directional* bottleneck arc, respectively. Note that the beamwidth θ_v at node v in (6) should be set as the smallest possible angle in the range between $\theta_{\min}$ and 2π to provide the beam-coverage for *all* nodes in $\Lambda_v^+(T_s)$. It has been proven in the recent literature that the Problem (3) belongs to P [12-14] and NP-hard [20] for networks with omni-directional antennas and directional antennas, respectively.

3 A New Distributed Algorithm

As mentioned earlier, the DMMT-DA algorithm [16] is one of the best solutions and especially beneficial to WANETs because of its distributed scheme. It runs in rounds to create a tree and each round includes as many nodes as possible on a minimum arc-weight (defined in Equation 4) basis until all nodes are in the tree. However, the following observation leads to the design of new heuristic algorithm that can improve the performance of DMMT-DA further.



(a) $T_1 = \{(s, a), (s, b), (s, c), (c, d)\}$ (b) $T_2 = \{(s, a), (s, b), (s, d), (d, c)\}$

Fig. 1. An example to show how the performance of DMMT-DA can be improved

A 5-node network instance is given in Fig. 1, in which source node s and all destination nodes a, b, c and d have the same energy level ε. Note that the Euclidean distance between each pair of nodes is exactly indicated in Fig. 1. We consider an intermediate solution obtained from the DMMT-DA algorithm with tree arcs (s, a) and (s, b). In the following iteration, we assume that arc (s, c) will be included into the tree by DMMT-DA because it has the minimum weight, *i.e.* $w_{sc} < w_{sd}$, or equivalently $p(r_{sc}, \angle asc) < p(r_{sd}, \angle asd)$, in which the symbol $\angle xyz$ denotes the angle between the two rays of yx and yz. Finally, the multicast tree T_1 is achieved by DMMT-DA as shown in Fig. 1a with $\delta_d(T_1) = p(r_{sb}, \angle asc) / \varepsilon$. Now we consider an alternative arc (s, d) to be included into the tree in the same iteration and the final tree should be T_2 as shown in Fig. 1b with $\delta_d(T_2) = p(r_{sb}, \angle asd) / \varepsilon$. It is obvious that T_2 is a better solution, *i.e.* $\delta_d(T_1) > \delta_d(T_2)$, because $\angle asc > \angle asc$. In other words, the solutions found by DMMT-DA based on the arc-weight may sometimes be far deviated from the optimum.

The above example motivates us to apply a node-centric approach, *i.e.* to use a node-weight instead of an arc-weight defined in (4) as the criteria, to increment a multicast tree such that the performance of DMMT-DA would be improved. In this section, we propose a new algorithm, DMMT-NC (Distributed Min-Max Tree algorithm with Node-Centric approach), for the min-max tree problem. The multicast tree is constructed in a distributed and incremental manner. Initially, the multicast tree T_s only contains the source node. It then iteratively performs a *Search-and-Grow* procedure until the tree contains all the nodes in M. The final multicast tree T_s is therefore obtained by pruning all transmissions that are not needed to reach the nodes in M.

We use T_s^i to denote an intermediate tree constructed by the DMMT-NC algorithm after the i-th node is added into the tree. As implied by the name of the algorithm, each node v maintains a node weigh w_v^i $(0 \le i \le n\text{-}1)$ at each step a tree is incremented, which is defined as follows.

$$w_v^i \equiv \min_{u\in N-N(T_s^i)} (w_{vu}^i) \tag{7}$$

$$w_{vu}^i \equiv p(r_{vu}^i, \varphi_{vu}^i) / \varepsilon_v.\, u \in N - N(T_s^i) \tag{8}$$

$$r_{vu}^i \equiv \max_{x\in\{u\}\cup\Lambda_v^+(T_s^i)} (r_{vx}) \tag{9}$$

$$\varphi_{vu}^i \equiv \min\left\{\theta_v \,\middle|\, \theta_v \text{ covers each node in } \{u\}\cup\Lambda_v^+(T_s^i)\right\} \tag{10}$$

Note that the variable r_{vu}^i denotes the longest Euclidean distance between node v and any node x already included in the tree T_s^i and a node u outside the tree. Similarly, variable φ_{vu}^i denotes the minimum beamwidth required by node v to cover all its child nodes already in the tree T_s^i as well as an additional node u outside the tree. In this way, the tree incremental operation by including the candidate node u, satisfying the condition

$$w_{vu}^i = w_v^i\,, \tag{11}$$

would lead to the lifetime of the resulting intermediate tree to be maximized over all possible choices of any node could be included into the tree. This approach is based on a node's point of view, which is different from other proposed algorithms.

In the following, we give a formal description of the DMMT-NC algorithm. The formulations shall help us understand the subsequent theoretical analysis that the proposed heuristic algorithm has an approximation ratio bounded by a constant number. The description of the DMMT-NC algorithm in pseudo code is given in Fig. 2.

The DMMT-NC Algorithm

```
Initialize $i = 0$, $N(T_s^i) = \{s\}$ and $A(T_s^i) = \phi$;
Repeat
    // Search Phase
    $\delta \equiv \min\{w_v^i \mid v \in N(T_s^i)\}$;
    // Grow Phase
    while $(\exists v \in N(T_s^i), w_v^i \le \delta \wedge \exists u \in N - N(T_s^i), w_{vu}^i = w_v^i)$
        $i = i + 1$;
        $N(T_s^i) = N(T_s^{i-1}) \cup \{u\}, A(T_s^i) = A(T_s^{i-1}) \cup \{(v,u)\}$;
        Update $w_v^i$ for each $v \in N(T_s^i)$ using (7 – 10);
until $(M \subseteq N(T_s))$;
Obtain the final multicast tree $T_s$ by pruning $T_s^i$.
```

Fig. 2. The DMMT-NC Algorithm

4 Theoretical Performance Analysis

In this section, we study the theoretical performance of the proposed algorithm in terms of approximation ratio[1]. We use δ_o^* and δ_d^* to denote the optimal solutions for the min-max tree problem under *omni-directional* and *directional* scenarios, respectively, *i.e.*

$$\delta_o^* = \min_{T_s \in \Omega_M} \delta_o(T_s) , \tag{12}$$

$$\delta_d^* = \min_{T_s \in \Omega_M} \delta_d(T_s) . \tag{13}$$

Given a multicast tree T_s obtained by the DMMT-NC algorithm, its approximation ratio ρ can be expressed as

$$\rho = \delta_d(T_s) / \delta_d^* . \tag{14}$$

In the following, we first provide several fundamental results that shall be used to derive the upper bound of the approximation ratio for the heuristic algorithm DMMT-NC. Let C_X denote the cut connecting a node partition X and N–X, in which the first node set X must include the source node s and the second node set $N - X$ must include at least one destination node, *i.e.*

$$C_X \equiv \{(v, u) \mid v \in X \wedge u \in N - X \wedge s \in X \wedge D \not\subset X\}. \tag{15}$$

We use $\psi(C_X)$ to denote the minimum weight of the cut links under omni-directional scenarios, *i.e.*

$$\psi(C_X) = \min_{(v,u)\in C_X} (p(r_{vu}, 2\pi) / \varepsilon_v) . \tag{16}$$

Theorem 1. If $G(N, A)$ is connected then for any cut C_X, then

$$\delta^* \geq \delta_0 \cdot \theta_{\min} / 2\pi . \tag{17}$$

Proof: Note that there is at least one destination node z ($z \in D$) belonging to $N - X$, *i.e.*, $z \in N - X$, because $D \not\subset X$. Let T_s^* be a min-max tree of network G with omni-directional antenna. There must exist an arc $(x, y) \in A(T_s^*)$ connecting X and $N - X$ (*i.e.*, $(x, y) \in C_X$) in order to satisfy that there must exist a directed path from s to the destination node z along the links in the tree T_s^*. Therefore, we can obtain (17) as follows. $\delta_o^* = \delta_0(T_s^*) = \max_{(v,u)\in A(T_s^*)} p(r_{vu}, 2\pi) / \varepsilon_v \geq p(r_{xy}, 2\pi) / \varepsilon_x \geq \min_{(v,u)\in C_X} p(r_{vu}, 2\pi) / \varepsilon_v = \psi(C_X)$. □

It is a straightforward exercise to obtain the following conclusion if a function $K_{vu}(\theta_1, \theta_2)$ is defined as

$$K_{vu}(\theta_1, \theta_2) \equiv p(r_{vu}, \theta_1) / p(r_{vu}, \theta_2) . \tag{18}$$

[1] An algorithm for a problem has an approximation ratio of $\rho(n)$ if, for any input of size n, the expected cost c of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost c^* of an optimal solution: $\max\{c / c^*, c^* / c\} \leq \rho(n)$.

Lemma 1. For any $(v, u) \in A$, $K_{vu}(\theta_1, \theta_2)$ satisfies

$$\begin{cases} \max(\theta_1 / \theta_2, p_{\min} / p_{\max}) \le K_{vu}(\theta_1, \theta_2) \le 1 & \theta_1 \le \theta_2 \\ 1 \le K_{vu}(\theta_1, \theta_2) \le \min(\theta_1 / \theta_2, p_{\max} / p_{\min}) & \theta_1 \ge \theta_2 \end{cases}. \tag{19}$$

Theorem 2. The optimal solutions δ_o^* and δ_d^* satisfy

$$\delta_d^* \ge \max(\theta_{\min} / 2\pi, p_{\min} / p_{\max}) \cdot \delta_o^* . \tag{20}$$

Proof: Considering $\theta_v \ge \theta_{\min}$ for any given multicast tree T_s and using Lemma 1, we then have the following derivations.

$$\begin{aligned} \delta_d^* &= \min_{T_s \in \Omega_M} \max_{(v,u) \in T_s} (p(r_{vu}, \theta_v) / \varepsilon_v) \\ &= \min_{T_s \in \Omega_M} \max_{(v,u) \in T_s} (K_{vu}(\theta_v, 2\pi) \cdot p(r_{vu}, 2\pi) / \varepsilon_v) \\ &\ge \min_{T_s \in \Omega_M} \max_{(v,u) \in T_s} (\max(\theta_v / 2\pi, p_{\min} / p_{\max}) \cdot p(r_{vu}, 2\pi) / \varepsilon_v) \\ &\ge \min_{T_s \in \Omega_M} \max_{(v,u) \in T_s} (\max(\theta_{\min} / 2\pi, p_{\min} / p_{\max}) \cdot p(r_{vu}, 2\pi) / \varepsilon_v) \\ &= \max(\theta_{\min} / 2\pi, p_{\min} / p_{\max}) \cdot \min_{T_s \in \Omega_M} \max_{(v,u) \in T_s} (p(r_{vu}, 2\pi) / \varepsilon_v) \\ &= \max(\theta_{\min} / 2\pi, p_{\min} / p_{\max}) \cdot \delta_o^* \end{aligned}$$

□

We now turn our attention to the most interesting and difficult task on deriving the approximation ratio of the DMMT-NC algorithm. Suppose that T_s is the final multicast tree obtained from the algorithm described in Fig. 2 and the *directional* bottleneck arc (v, u) of T_s is the i-th arc added into the tree, *i.e.* the intermediate tree is T_s^{i+1} after arc (v, u) is included. Let φ_v be the beamwidth applied by the node v in T_s. The solution $\delta_d(T_s)$ can thus be expressed as follows.

$$\begin{aligned} \delta_d(T_s) &= p(r_{vu}, \varphi_v) / \varepsilon_v \\ &= K_{vu}(\varphi_v, \varphi_{vu}^i) \cdot p(r_{vu}, \varphi_{vu}^i) / \varepsilon_v \\ &\le K_{vu}(\varphi_v, \varphi_{vu}^i) \cdot p(r_{vu}^i, \varphi_{vu}^i) / \varepsilon_v \; . \\ &= K_{vu}(\varphi_v, \varphi_{vu}^i) \cdot w_{vu}^i \\ &= K_{vu}(\varphi_v, \varphi_{vu}^i) \cdot w_v^i \end{aligned} \tag{21}$$

We further assume that arc (v', u') is the first one added into the tree in the same round of the *Search-and-Grow* iteration (described in Fig. 2) as arc (v, u) is included and the resulting intermediate tree is T_s^{j+1} ($j \le i$). Based on the description in Fig. 2, we have $w_v^i \le w_{v'}^j$. Now we define a cut C_X, where $X \equiv N(T_s^j)$, and let arc (x, y) be the one in C_X such that

$$\psi(C_X) = p(r_{xy}, 2\pi) / \varepsilon_x . \tag{22}$$

Recall that just before the intermediate tree T_s^{j+1} is formed, arc (v', u'), instead of (x, y), is chosen to be included, which implies $w_{v'}^j \le w_x^j$. By summarizing the above analysis, we have the derivation as follows.

$$w_v^i \le w_{v'}^j \le w_x^j \le w_{xy}^j = p(r_{xy}^j, \varphi_{xy}^j) / \varepsilon_x \tag{23}$$

We consider the following two cases.

Case 1: $r_{xy}^j = r_{xy}$

The above equation can be rewritten as

$$w_v^i \le p(r_{xy}, \varphi_{xy}^j) / \varepsilon_x = K_{xy}(\varphi_{xy}^j, 2\pi) \cdot p(r_{xy}, 2\pi) / \varepsilon_x \, . \tag{24}$$

Case 2: $r_{xy}^j = r_{xz_1}$ as shown in Fig. 3.

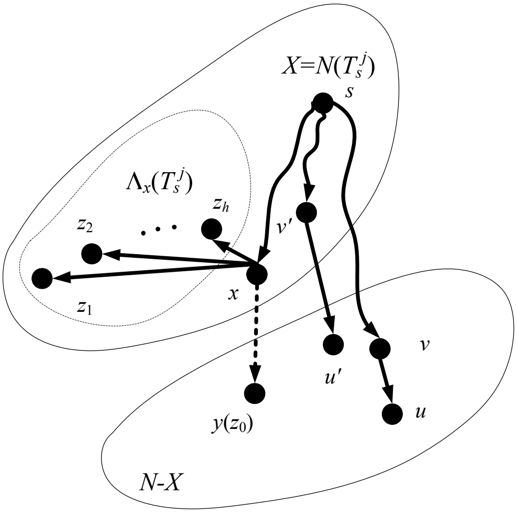


Fig. 3. Illustration used to derive the approximation ratio of DMMT-NC

This means arc (x, z_1) is already in the tree T_s^j and we assume the resulting tree is $T_s^{k_1}$ ($k_1 < j$) just after it is included. Considering arc (x, z_1), instead of (x, y), is chosen to be included at that moment, we have $w_{xz_1}^{k_1} \le w_{xy}^{k_1}$ or equivalently

$$p(r_{xz_1}^{k_1}, \varphi_{xz_1}^{k_1}) \le p(r_{xy}^{k_1}, \varphi_{xy}^{k_1}) \, . \tag{25}$$

Furthermore, the condition $r_{xy}^j = r_{xz_1}$ also implies

$$r_{xz_1}^{k_1} = r_{xz_1} \, . \tag{26}$$

Now equation (23) under case 2 can be rewritten as follows by combining (25) and (26).

$$\begin{aligned} w_v^i &\le p(r_{xz_1}, \varphi_{xy}^j) / \varepsilon_x \\ &= K_{xz_1}(\varphi_{xy}^j, \varphi_{xz_1}^{k_1}) \cdot p(r_{xz_1}, \varphi_{xz_1}^{k_1}) / \varepsilon_x \\ &= K_{xz_1}(\varphi_{xy}^j, \varphi_{xz_1}^{k_1}) \cdot p(r_{xz_1}^{k_1}, \varphi_{xz_1}^{k_1}) / \varepsilon_x \\ &\le K_{xz_1}(\varphi_{xy}^j, \varphi_{xz_1}^{k_1}) \cdot p(r_{xy}^{k_1}, \varphi_{xy}^{k_1}) / \varepsilon_x \end{aligned} \tag{27}$$

Comparing (27) and (23), we can conclude that the above equation can be further derived similarly under two cases of 1) $r_{xy}^{k_1} = r_{xy}$ or 2) $r_{xy}^{k_1} = r_{xz_2}$ as shown in Fig. 3 until Case 1) is met.

Generally, we assume that the Case 1) is met at the *h*-round of the above derivation iteration, *i.e.*

$$\begin{cases} r_{xy}^{k_l} = r_{xz_{l+1}} & 0 \le l \le h-1 \\ r_{xy}^{k_l} = r_{xy} & l = h \end{cases}, \tag{28}$$

and the following equation will be eventually achieved.

$$\begin{aligned} w_v^i &\le \prod_{l=1}^{h} K_{xz_l}(\varphi_{xz_{l-1}}^{k_{l-1}}, \varphi_{xz_l}^{k_l}) \cdot p(r_{xy}^{k_h}, \varphi_{xy}^{k_h}) / \varepsilon_x \\ &= \prod_{l=1}^{h} K_{xz_l}(\varphi_{xz_{l-1}}^{k_{l-1}}, \varphi_{xz_l}^{k_l}) \cdot p(r_{xy}, \varphi_{xy}^{k_h}) / \varepsilon_x \\ &= \prod_{l=1}^{h} K_{xz_l}(\varphi_{xz_{l-1}}^{k_{l-1}}, \varphi_{xz_l}^{k_l}) \cdot K_{xy}(\varphi_{xy}^{k_h}, 2\pi) \cdot p(r_{xy}, 2\pi) / \varepsilon_x \\ &= H \cdot p(r_{xy}, 2\pi) / \varepsilon_x \end{aligned} \tag{29}$$

Note that item *H* in (29) is defined as

$$H \equiv K_{xy}(\varphi_{xy}^{k_h}, 2\pi) \cdot \prod_{l=1}^{h} K_{xz_l}(\varphi_{xz_{l-1}}^{k_{l-1}}, \varphi_{xz_l}^{k_l}) \tag{30}$$

and the boundary conditions of (29) are given below.

$$k_0 = j, z_0 = y, 0 \le h \le \lambda_x^+(T_s^j) \tag{31}$$

Finally, combining (21), (29), (22), (17) and (20) sequentially, we obtain

$$\begin{aligned} \delta_d(T_s) &\le K_{vu}(\varphi_v, \varphi_{vu}^i) \cdot w_v^i \\ &\le K_{vu}(\varphi_v, \varphi_{vu}^i) \cdot H \cdot p(r_{xy}, 2\pi) / \varepsilon_x \\ &= K_{vu}(\varphi_v, \varphi_{vu}^i) \cdot H \cdot \psi(C_X) \\ &\le K_{vu}(\varphi_v, \varphi_{vu}^i) \cdot H \cdot \delta_o^* \\ &\le K_{vu}(\varphi_v, \varphi_{vu}^i) \cdot H \cdot \delta_d^* / \max(\theta_{\min} / 2\pi, p_{\min} / p_{\max}). \end{aligned} \tag{32}$$

The above analysis now allows us to obtain the following conclusion.

Theorem 3. The DMMT-NC algorithm is a constant-factor approximation algorithm with an approximation ratio ρ bounded by

$$\rho \le \mu_\rho \equiv H \cdot K_{vu}(\varphi_v, \varphi_{vu}^i) \cdot \min(2\pi / \theta_{\min}, p_{\max} / p_{\min}) . \tag{33}$$

It is a straightforward exercise based on (19) to verify that μ_ρ is bounded by a constant number. In particular, we can conclude $H \le 1$ since

$$\varphi_{xz_{l-1}}^{k_{l-1}} \le \varphi_{xz_l}^{k_l} \le 2\pi, \quad 1 \le l \le h . \tag{34}$$

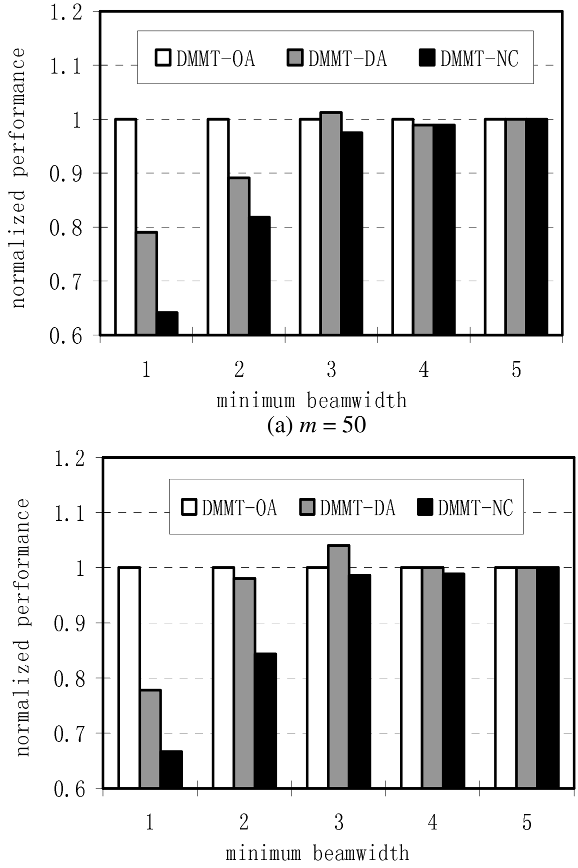
On the other hand, it is not sure $K_{vu}(\varphi_v, \varphi_{vu}^i) \le 1$ because the relation of φ_v and φ_{vu}^i is not deterministic in general due to the post-pruning operation.

5 Experimental Performance Evaluation

We have performed a simulation study for evaluating a set of distributed algorithms DMMT-OA [16], DMMT-DA [16] and the new proposed DMMT-NC. Their solutions

Table 1. Parameter values for simulation

Parameters	Values
n	100
m	50 and 100
$\theta_{\min}$	15°, 30°, 60°, 90°, and 360°
$p_{\max}$	10
$p_{\min}$	0.1
$(E(\varepsilon), D(\varepsilon))$	(500, 200)
α	2



(a) $m = 50$

(b) $m = 100$

Fig. 4. Normalized performance as a function of the minumum beamwidths 15°, 30°, 60°, 90°, and 360° (corsponding to the numbers 1 – 5 on the x-axle) under various multicast sizes

are denoted as δ_1, δ_2 and δ_3, respectively. We use the metric δ_i/δ_1 ($i = 1, 2, 3$) to evaluate their relative performance, which allows us to facilitate the comparison of different algorithms over a wide range of network examples. In each network example, a number of nodes are randomly generated within a square region 10 × 10. The values of parameters used in simulation are given in Table 1. We randomly generated 100 different network examples, and we present here the average over those examples for all cases.

Fig. 4 depicts graphically the normalized performances over different connected network topologies. The x-axis represents the minimum beamwidths 15°, 30°, 60°, 90° and 360° (corresponding to the numbers 1 - 5 on the x-axle) and the y-axis presents the mean of δ_i/δ_1 for all three distributed algorithms. Referring to the multicast size $m = 50$ in Fig. 4a, we observe that the new proposed distributed algorithm DMMT-NC improves the other two algorithms significantly when the minimum beamwidth is small. In particular, such improvement is over 30% and 15% compared to DMMT-OA and DMMT-DA, respectively. On the other hand, once the minimum beamwidth increases (greater than 90°), all algorithms converge to the same performance (optimal solutions [16]). A similar observation can be made for the broadcasting scenarios $m = 100$ as shown in Fig. 4b.

6 Conclusion

We have presented a new distributed long-lived multicast algorithm for directional communications in wireless ad hoc networks. Our proofs show that it is a constant-factor approximation algorithm. Our efforts are also validated via the simulation study, in which the experimental results show that our new algorithm has better performance than other distributed algorithms.

References

1. Wieselthier, J.E., Nguyen, G.D., et al.: On the Construction of Energy-Efficient Broadcast and Multicast Trees in Wireless Networks. In: IEEE INFOCOM, pp. 585–594 (2000)
2. Guo, S., Yang, O.: A Dynamic Multicast Tree Reconstruction Algorithm for Minimum-Energy Multicasting in Wireless Ad Hoc Networks. In: IEEE IPCCC, pp. 637–642 (2004)
3. Cartigny, J., Simplot, D., Stojmenovic, I.: Localized minimum-energy broadcasting in ad-hoc networks. In: IEEE INFOCOM, pp. 2210–2217 (2003)
4. Wan, P.J., Calinescu, G., et al.: Minimum-energy broadcast routing in static ad hoc wireless networks. In: IEEE INFOCOM, pp. 1162–1171 (2001)
5. Wan, P.J., Yi, C.W.: Minimum-Power Multicast Routing in Static Ad Hoc Wireless Networks. IEEE/ACM Transactions on Networking 12(3), 507–514
6. Cagalj, M., Hubaux, J.-P., Enz, C.: Minimum-energy broadcast in all-wireless networks: NP-completeness and distribution issues. In: ACM Mobicom, pp. 172–182 (2002)
7. Kang, I., Poovendran, R.: On the Lifetime Extension of Energy-Efficient Multihop Broadcast Networks, the World Congress on Computational Intelligence (2002)
8. Kang, I., Poovendran, R.: Maximizing Static Network Lifetime of Wireless Broadcast Ad-hoc Networks. In: IEEE ICC, pp. 2256–2261 (2003)

9. Das, A.K., Marks II, R.J., El-Sharkawi, M.A., Arabshahi, P., Gray, A.: MDLT: a polynomial time optimal algorithm for maximization of time-to-first-failure in energy-constrained broadcast wireless networks. In: IEEE Globecom, pp. 362–366 (2003)
10. Cheng, M.X., Sun, J., et al.: Energy-efficient Broadcast and Multicast Routing in Ad Hoc Wireless Networks. In: IEEE IPCCC, Phoenix, Arizona, April 2003, pp. 87–94 (2003)
11. Wang, B., Gupta, S.K.S.: On Maximizing Lifetime of Multicast Trees in Wireless Ad hoc Networks. In: International Conference on Parallel Processing, pp. 333–340 (2003)
12. Floréen, B., Kaski, P., et al.: Multicast time maximization in energy constrained wireless networks. In: Workshop on Foundations of Mobile Computing, pp. 50–58 (2003)
13. Georgiadis, L.: Bottleneck multicast trees in linear time. IEEE Communications Letters 7(11), 564–566 (2003)
14. Guo, S., Leung, V., Yang, O.: A Scalable Distributed Multicast Algorithm for Lifetime Maximization in Large-scale Resource-limited Multihop Wireless Networks. In: ACM IWCMC, pp. 419–424 (2006)
15. Guo, S., Yang, O.: Multicast Lifetime Maximization for Energy-Constrained Wireless Adhoc Networks with Directional Antennas. In: IEEE Globecom, pp. 4120–4124 (2004)
16. Guo, S., Leung, V., Yang, O.: Distributed Multicast Algorithms for Lifetime Maximization in Wireless Ad Hoc Networks with Omni-directional and Directional Antennas. In: IEEE Globecom (2006)
17. Wieselthier, J.E., Nguyen, G.D., Ephremides, A.: Energy-Aware Wireless Networking with Directional Antennas: The Case of Session-Based Broadcasting and Multicasting. IEEE Transactions on Mobile Computing 1(3), 176–191 (2002)
18. Wieselthier, J.E., Nguyen, G.D., et al.: Energy-Limited Wireless Networking with Directional Antennas: The Case of Session-Based Multicasting. In: IEEE INFOCOM, pp. 190–199 (2002)
19. Guo, S., Yang, O.: Optimal Tree Construction for Maximum Lifetime Multicasting in Wireless Ad-hoc Networks with Adaptive Antennas. In: IEEE ICC, pp. 3370–3374 (2005)
20. Hou, Y., Shi, Y., Sherali, H.D., Wieselthier, J.E.: Online lifetime-centric multicast routing for ad hoc networks with directional antennas. In: IEEE INFOCOM, pp. 761–772 (2005)

Maintaining Quality of Service with Dynamic Fault Tolerance in Fat-Trees

Frank Olaf Sem-Jacobsen[1,2] and Tor Skeie[1,2]

[1] Department of Informatics
University of Oslo
Oslo, Norway
[2] Networks and Distributed Systems
Simula Research Laboratory
Lysaker, Norway

Abstract. A very important ingredient in the computing landscape is Utility Computing Data Centres (UCDCs), large-scale computing systems that offer computational services to concurrently running jobs through virtual servers. As UCDC systems increase in size and the mean time between failure decreases, it is becoming an increasingly important challenge to expediently tolerate failures (dynamically), while distributing the effects of the failure amongst the virtual servers according to their service level agreements. We propose and evaluate a strategy for offering predictable service in fat-trees experiencing faults, by reprioritising packets. The strategy is able to distribute the effect of network faults in order to satisfy a number of quality-of-service demands. Which demands to favour depends on the computer system and the characteristics of the jobs it is running, and in the presence of a moderate number of faults it is to some degree possible to meet the demands.

1 Introduction

The application of supercomputer systems is increasing. Traditionally there are the "single job" supercomputers, where single jobs run exclusively for a certain amount of time. More recently, we see the emergence of Utility Computing Data Centres (UCDC), where multiple jobs are run in parallel on the same supercomputer, separated into virtual servers. This brings forth the necessity of being able to partition, or virtualise, the supercomputer in order to separate the different jobs from each other. In this manner, each job may run on a dedicated set of resources without interfering with other jobs, and receive predictable service. Typically, many network resources must be shared between the jobs. To provide quality of service some kind of differentiating between the jobs/virtual servers must be undertaken relative to existing Service Level Agreements (SLA). This places severe demands on job scheduling and resource allocation, but also on the interconnection network and the routing algorithm used to direct packets through the system. The network must be able to reliably forward packets such that each job is guaranteed a portion of the capacity in the network (quality of

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 451–464, 2008.

service), even if parts of the network should cease to function for any period of time (fault tolerance).

As opposed to quality of service which usually is built into the network technology using virtual channels (VC), fault tolerance is often considered an “add-on” mechanism because it may be achieved by having an appropriate routing algorithm to reconfigure the network to restore connectivity on fault events, independently of the interconnect technology. Alternatively, the switches adjacent to the fault may dynamically reroute packets around the fault using preconfigured paths. This requires that the fault tolerance mechanism is more closely integrated to the network technology.

Reconfiguration and rerouting affects traffic in the network in different ways. While reconfiguration affects the entire network and separates the configurations either in time (halting and draining the network) or space (moving the traffic to a different set of virtual channels) and thus affecting all traffic in the network, dynamic rerouting affects only a subset of the traffic, perhaps only the path of a single flow is changed because of a fault.

Although many interconnection networks employ both fault-tolerance and quality-of-service mechanisms, no work has been done on combining the two system demands. A reason for this might be that fault events are rare when compared to the lifetime of network flows. Also, static reconfiguration need not necessarily consider quality-of-service issues since the entire network, or most of the network, is reconfigured, thus changing the conditions for all traffic in the network. However, as interconnection networks are used in high-speed systems that support a heterogeneous set of jobs, as is the case with UCDCs, there is a need for local dynamic rerouting algorithms which are able to quickly tolerate the fault while losing as few packets as possible. At the same time, it is important that traffic in the network that is not affected by the fault maintains its perceived quality of service, while the traffic affected by the fault receives as good service as possible without degrading the service of other traffic. It is therefore necessary to make the dynamic fault tolerance mechanism QoS-aware.

The purpose of this paper is to propose and evaluate a strategy for maintaining quality of service for flows in the network, both for flows that are affected and unaffected by the fault, while using a local dynamic rerouting algorithm which routes packets around network faults locally. We discuss in what ways flows are affected by faults, and how this effect can be most efficiently distributed in the system. We then evaluate the ability of the strategy to satisfy the various quality of service requirements put forth by UCDC systems, namely ensuring high network utilisation, isolating the effect of faults to the directly affected flows, and preserving quality of high-priority traffic, if necessary at the expense of low-priority traffic. We also consider the requirements of single-job systems and how these too may be satisfied. In this paper we focus on the fat-tree topology [8], since this is a widely used topology for interconnection networks employed in UCDCs. For instance, the fat-tree is an integral part of several interconnect technologies such as Infiniband, Myrinet [5] and QsNet/QsNetII [3].

The rest of this paper is organised as follows. We first give an overview of previous work in the field of dynamic rerouting fault tolerance algorithms and quality of service in Section 2. We then describe the local dynamic rerouting algorithm we will make QoS-aware in Section 3, present our strategy for achieving this in Section 4, and discuss the targets for our quality of service mechanism in Section 5. The strategy is evaluated in Section 6 and the paper is concluded in Section 7.

2 Previous Work

Both fault tolerance and quality of service have received much attention from the academic world. With regards to fault tolerance, much work has been done on improving the fault tolerance of existing network topologies. This is achieved through adding extra hardware in terms of switches and links [19], routing the packets through the network in multiple passes [4,7], or combining the two approaches [18].

The above approaches only provide reconfiguration or endpoint dynamic rerouting. However, similar techniques may be used to create network topologies supporting local dynamic rerouting. By adding additional links and switches, several MIN topologies supporting local dynamic rerouting have been created, e.g. the Quad Tree [16], a modified Omega network [17], and the Siamese-twin fat-tree [15].

Recently, a local dynamic rerouting algorithm for fat-trees has been developed, both for adaptive [13], deterministic [12], and source routed fat-trees [11]. This is well suited for our purpose and is the local dynamic fault-tolerant routing algorithm we employ in this paper. We describe this algorithm in greater detail in the next section.

Concerning quality of service, much work has been done on exploring the possibilities of the quality of service mechanisms provided by Infiniband. The core of the quality-of-service mechanism in Infiniband is the arbitration tables that dictate how much of the total link bandwidth each virtual channel may receive. Alfaro et al. [1] propose mechanisms to compute these arbitration tables based on bandwidth requirements, and show how the tables may be configured to serve time sensitive traffic [2]. This has also been done for advanced switching [9].

Although much work has been done in the two fields of quality of service and fault tolerance separately, to the best of the authors knowledge, no work has been published considering the effect of fault tolerance mechanisms on quality of service.

3 Dynamic Fault Tolerance

Before we go into the possible methods of maintaining quality of service with network faults, we present the dynamic fault tolerant routing algorithm we will be using and the topology of choice. We use a recently developed routing algorithm

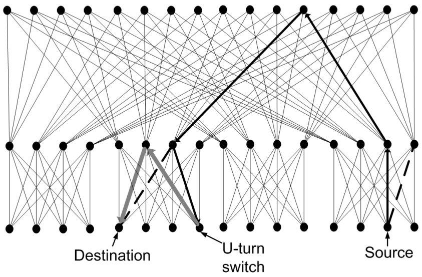


Fig. 1. A fat-tree (4-ary 3-tree) consisting of radix 8 switches with two link faults. The faulty link are marked as bold, dashed lines, and the bold line describes the path of a packet from its source to its destination. Note how the packet is misrouted via the U-turn switch and that the two subsequent links are grey, indicating that the packet is in the deadlock freedom channels.

that provides local dynamic rerouting (routing around faulty elements) for fat-trees [12].

The fat-tree is a tree topology, often realised as a k-ary n-tree [10], where k is the number of switch ports in the upward or downward direction, and n is the number of switching stages (Figure 1).

The first version of the dynamic rerouting algorithm for fat-trees required k virtual channels to tolerate $k-1$ link or switch faults. The deadlock-freedom proof of the algorithm has been further refined to show that only a single additional virtual channel is required for $k-1$ fault tolerance [14]. We give a brief outline of the most important properties of the algorithm, for further details we refer to [14]. An example of the operation of the algorithm is provided in Figure 1.

Packet forwarding in fat-trees is divided into two phases, upwards and downwards. Packets forwarded in the k-ary n-tree may use any of the upward links to advance towards their destination in the upward phase, but in the downward phase there is only a single deterministic path, determined by the path chosen in the upward phase. Thus, achieving local dynamic rerouting around link faults in the upward phase is trivial: if the original upward link is faulty, simply choose another upward link. In the downward phase we must resort to non-minimal paths if the link towards the destination has failed. A packet encountering a faulty link in the downward phase must choose an alternative downward link that does not lead towards its destination, to what becomes a *U-turn Switch.* Once this link is traversed, normal shortest path routing may commence, first directing the packet upwards one stage from the u-turn switch over a different link from where it arrived, and then downwards. If this downward path is also faulty the packet is returned to the U-turn switch, which may select a different upward link. The upward and downward path following the U-turn switch must take place in a deadlock freedom layer, an additional virtual layer (a specific virtual channel on each link) in the network, to ensure deadlock freedom.

The path of a packet that encounters a link fault, first in the upward phase, and then in the downward phase with subsequent rerouting, is displayed in

Figure 1. The bold, dashed links represents faulty links on the original path, and the unbroken bold links are the actual path of the packet when avoiding the faults. Note that over the two subsequent grey-coloured links from the U-turn switch, the packet is forwarded in the deadlock freedom layer.

In the next section will explore how quality-of-service guarantees may be maintained when using this algorithm to tolerate multiple link faults.

4 Maintaining Quality of Service with Dynamic Fault Tolerance

Given the routing algorithm presented in the previous section, let us identify at which point it is possible, or necessary, to consider quality-of-service requirements. Even though we focus on the routing algorithm we just presented, the strategies we propose and the results we gather are applicable to other routing algorithms. Specifically, routing algorithms that rely on extra virtual channels to ensure deadlock freedom will have to take our results into consideration. We assume that each link in the network is configured with two or more virtual channels of different priority, i.e. virtual channels with higher priority are guaranteed a larger portion of the link bandwidth.

This is often achieved by assigning a weight to each virtual channel corresponding to the fraction of the link bandwidth assigned to that channel. An arbitrator can use the weight of each virtual channel to determine how many packets it may transmit when selected.Virtual channels are usually served in a round-robin fashion.

Each time a packet encounters a faulty link on its path through the network, it will be misrouted and thus deviate from its original path. Packets that encounter faults in the upward phase will follow a different path through large parts of the network, while packets that encounter a fault in the downward phase will cross three links not part of their original path. In other words, packets that encounter faults will add load to other parts of the network, thereby interfering with traffic not directly affected by the fault. This shifting of load may degrade the QoS experienced by the traffic being misrouted. More importantly, the rerouted traffic may also degrade the QoS provided to the traffic already on the paths to which it is rerouted. The added load will decrease the achieved throughput of the individual flows, and increase the latency because of increased queue lengths.

There is one basic way of handling this misrouted traffic with respect to quality of service, namely change its priority. Recall that when using virtual channels, changing the priority of a packet corresponds to shifting the packet to a virtual channel with a different weight in the arbitration tables. However, what priority to change to, and when/if to change back to the original priority gives rise to a large number of possible approaches.

There are three events for which it is natural to consider changing the priority of a packet.

1. The packet encounters a fault and has to be rerouted.
   (a) In the upward phase.
   (b) In the downward phase.
2. The rerouted path merges with the original fault-free path.
3. The packet is switched to the deadlock freedom channels after the U-turn.

Event 1 (a and b) is the obvious event for which it might beneficial to change the priority of a packet to minimise the impact of the packet following a different path. Similarly, event 2 is the case where the packet has finished the rerouting operation and rerouted path merges with the path the packet would have followed had there not been any fault. Whereas changing the priority of packets at the two first events is optional, packets that encounter faults in the downward phase must necessarily be switched to a deadlock-freedom channel in the U-turn switch. As for the other VCs, these virtual channels also must have a weight in the arbitration table, and thus a priority. Consequently, packets that encounter faults in the downward phase will all be forwarded through at least two virtual channels of the same priority x, regardless of whether they are high ($> x$) or low-priority ($< x$) packets in the first place. The priority given to these deadlock freedom channels will therefore have a significant impact on traffic of all priorities. This is further aggravated by the fact that interconnection networks rely on link level flow control with a back-pressure mechanism to ensure that no uncorrupted packets are lost. Any slowdown or speed-up of the misrouted packets will consequently affect all packets upstream from the point where the change occurs.

In the next section we discuss how the effects of link faults should be distributed in the system, and which combinations of the different alternatives may yield the desired results.

5 Managing Quality of Service in Supercomputers and UCDCs

The optimal way of distributing the effect of network faults differs depending on the application of the supercomputer. For supercomputers which traditionally run a single job at a time, the entire interconnection network is used by the job, with some small amount of management traffic.Thus, it is clear that any mechanism to reprioritise packets that are misrouted around a fault should be designed to maximise network efficiency.

The situation is much more complex in UCDCs since there are several jobs competing for the same network resources. We assume that jobs are allocated using virtual channels to achieve separation between the various jobs/virtual servers. Each VC may be assigned a priority to give certain jobs higher bandwidth/lower latency access to key resources, for instance, based on how much the customers have paid to run the job (manifested in a SLA). In this context, the best way of distributing the effect of network faults is quite different from the single-job case. Even though it is beneficial for the system as a whole to maximise in efficiency of the entire network, the consideration for the different jobs and their SLAs plays an important role. High-priority jobs should receive maximum priority around the faults at the expense of low-priority jobs. On the other hand, it could be argued that faults occurring in a part of the network primarily used by a single job should only affect that single job, without degrading the service of other jobs. However, this may not result in the best overall performance.

The question of how quality of service should be handled when misrouting around network faults becomes a question of how one wishes to distribute the effects of the fault in the network, i.e. slow down a single job significantly versus slowing down multiple jobs, but where each job is only marginally affected.

We now discuss the possible strategies given the available mechanism of changing the priority of misrouted traffic. For simplicity, we assume that we have three priority levels, one high-priority level, one medium-priority level, and one low-priority level. In a UCDC context we may assume that the high-priority level is used by a high-priority job, and the medium and low-priority levels are used by a medium and low-priority job respectively. Each of these three levels are mapped to their own VC on every link. Additionally, there are the deadlock freedom channels which also have a specific priority/weight. The exact value of this weight varies depending on the various approaches we evaluated in the next section.

This allows us to propose numerous approaches that can be broadly divided into two main categories, namely one where misrouted traffic is given high priority and another where misrouted traffic is given low-priority. Within these two categories there are multiple alternatives for what is done with the packet after being misrouted and the priority of the deadlock-freedom channel.

Decreasing the priority of the misrouted traffic is intuitively a good approach. By decreasing the priority of the traffic being misrouted, its impact on other traffic in the network not affected by the fault is minimal. However, as we see in the next section, the back-pressure nature of interconnection networks will cause this reprioritised traffic to severely impact other traffic in the network. On the other hand, increasing the priority of misrouted traffic will ensure that it is expediently handled and possibly make up for the fact that it has a longer path to travel in the presence of the fault, but it might cause a degradation of the service to high-priority traffic.

If we generate all combinations of increasing and lowering packet priority when misrouting upwards and downwards, as well as having high and low-priority deadlock freedom channels and returning to the original priority after the misrouting is complete or continuing to the destination with the misroute priority, we get 14 possible combinations. In the next section we present and evaluate these 14 different combinations and discuss which one may be most suited to fulfill quality of service demands on the basis of what we have discussed here.

6 Evaluation

To evaluate the behaviour of network traffic of different traffic classes in the presence of link faults we have performed an extensive set of simulations.

6.1 Simulation Parameters

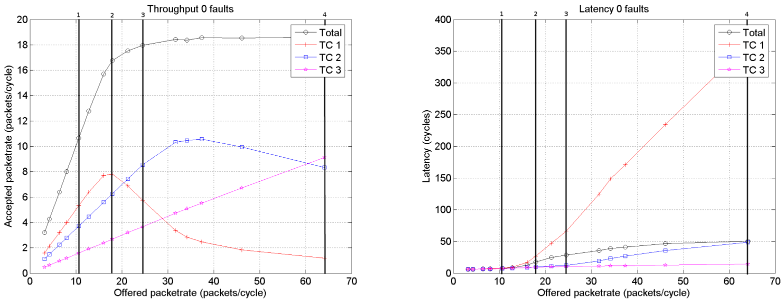
The simulations are performed in a simulator based upon j-sim [6] and developed in-house at Simula Research Laboratory. The network is configured with three traffic classes (TC1, TC2, TC3), each assigned to a virtual channel in the network corresponding to low (VC1), medium (VC2), and high (VC3) priority traffic. TC1 is assigned 40% of the total traffic, TC2 is assigned 35%, and TC3 is assigned 25% of the total traffic offered to the network. Additionally, there is a fourth virtual channel (VC4) for use for deadlock freedom when misrouting around link faults, which will alternately have the same priority as VC3 or just above VC1 for the different scenarios. The fat-tree topology of choice for the simulations is a 4-ary 3-tree, consisting of switches of radix eight, interconnected in three tiers. This is sufficiently small to allow the simulations to be terminated within a reasonable time. Furthermore, when scaling to large network sizes, although the relative differences of the different approaches may decrease, the overall conclusions will remain the same.

Traffic is generated following a Poisson distribution. The destination address distribution is such that all paths are of equal length when there are no link faults, i.e. all possible destinations for any given source lies in the other half of the network, forcing all traffic through the top stage switches. The network is allowed to stabilise before statistics are recorded and faults are introduced. Thereafter the simulations are run for 10 000 simulation cycles.

The relevant setup for the 14 different alternatives is summarised in Table 1. $VC2-4$ gives the percent of bandwidth reserved for traffic in the respective VCs (VC1 allways has a weight of 1, and is therefore omitted), VC_m determines which VC the packet is moved to when it is misrouted and "Far" describes whether the packet is returned to its original VC after it joins its original path (local), or if it retains the new VC until the endpoint (to end). U and D indicate whether

Table 1. A list of the different experiments run

S	VC2	VC3	VC4	VC_m	Far	U	D
1	35	54	10	VC1	local	X	X
2	35	54	10		local		
3	35	54	10	VC3	local	X	X
4	20	40	39		local		
5	20	40	39	VC1	local	X	X
6	20	40	39	VC3	local	X	X
7	35	54	10	VC1	to end	X	X
8	35	54	10	VC3	to end	X	X
9	20	40	39	VC1	to end	X	X
10	20	40	39	VC3	to end	X	X
11	35	54	10	VC1	local		X
12	35	54	10	VC3	local		X
13	20	40	39	VC1	local		X
14	20	40	39	VC3	local		X



(a) Throughput

(b) Latency

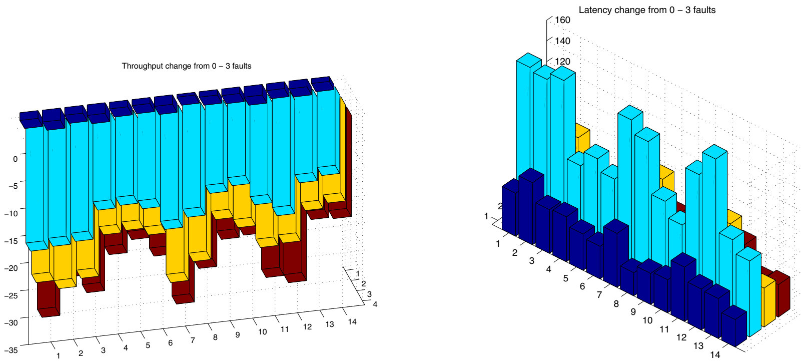
Fig. 2. Throughput of the network without faults

the packet changes VC when it encounters a fault in the upward and downward directions respectively.

6.2 Simulation Results

Following from Section 3, the 4-ary 3-tree is guaranteed to be connected with up to and including three link faults ($k = 4$). In this evaluation we present the difference in throughput and latency for the traffic classes when comparing a fault free network to a network with three link faults. This comparison is done for 4 different load scenarios corresponding to the vertical lines in Figure 2. Figure 2a) shows the throughput (y-axis) of the three traffic classes in the fault free network for an increasing traffic injection rate (x-axis). Similarly, Figure 2b) shows the latency (y-axis) for the same simulations. The 4 vertical lines mark the load cases selected for detailed analysis. The figures clearly show how the throughput of the lower priority traffic classes diminishes as the network saturates in favour of the high-priority traffic class TC3.

Figures 3 a) and b) show the total throughput and latency respectively for the 14 test cases presented above, and the four load cases marked in the previous figure. Every bar in the plot indicates the reduction in throughput when comparing the network with three link faults against the fault free network. The bars closest to us represents load case 1, while the bars furthest back represents load case 4. Note that the base of the bars is at zero at the top, and they stretch downwards indicating the throughput reduction in percent. For load case 1, the throughput reduction is insignificant, but as the load increases, the impact of link faults becomes more pronounced. When looking at either of the load cases it is clear that the approaches 4, 5, 6, 9, 10, 13, and 14 give similar results and the smallest throughput reduction. The common factor of these approaches is that the deadlock-freedom channel, VC4, has the same priority as the high-priority channel VC3. Within this set, approach 10 provides the smallest reduction for load case 2, while approach 5 gives the smallest reduction for load cases 3 and 4 (the two saturated cases). Approach 10 gives misrouted packets high



(a) Throughput

(b) Latency

Fig. 3. Change in throughput and latency from 0-3 faults for 4 loads

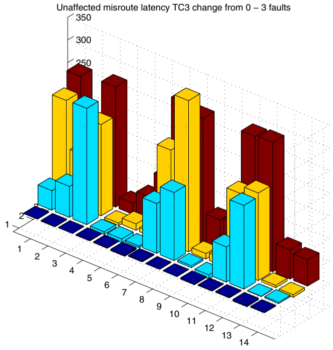


Fig. 4. Change in latency for unaffected traffic in TC3

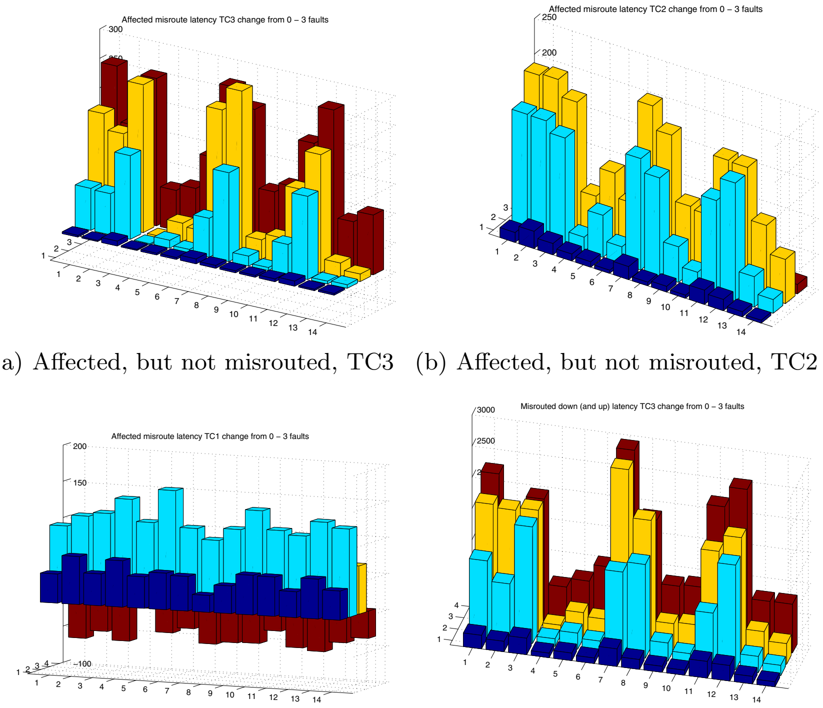
priority from the moment they encounter any fault, either upwards or downwards, until they reach their destination. As we see later, this affects the performance of other high-priority traffic in the network. The other highest performing approach, approach 5, switches misrouted packets to the low-priority VC1 both when encountering fault upwards and downwards, but they are returned to their original VC when they return to their original path. In this manner, traffic deviating from its original path in any way is given the lowest priority (except when in the deadlock freedom channels) allowing the highest utilisation of the network overall.

The throughput results are mirrored in the latency plot where the approaches utilising a high-priority deadlock-freedom channel provide the smallest increase in latency. Note that the bars go from the bottom upwards indicating a latency increase given in percent on the y-axis.

Figure 4 shows the change in latency for high-priority TC3 traffic which does not encounter any faults and does not share links with any other traffic encountering faults. As before, and for all remaining plots, load case 1 is the frontmost row of bars, while load case 4 is the row of bars furthest back. The y-axis is the change in packet latency in percent. The latency for this traffic should ideally not show any change when faults are introduced, as this traffic should be unaffected. However, the figure clearly shows that, for all approaches, the traffic experiences a latency increase. To explain this, recall that interconnection networks employ link level flow control, so should any packet be slowed at any point in the network (due to head of line blocking, etc.), all upstream traffic from this packet using the same VC will to some degree be affected through having to wait longer in some buffers, thus increasing the latency. Hence, even though this traffic does not share any links with packets that are misrouted along their path, they will share links with packets that share links with packets directly affected by the fault, creating a complex indirect dependency chain. This effect is clearly visible in the figure. It shows the same distinction between the approaches using a high-priority deadlock freedom channel and those using a low-priority deadlock freedom channel as we saw for the overall throughput. It is fairly obvious that ensuring expedient handling of misrouted traffic by switching it to high-priority channels is important to be able to maintain high-priority guarantees. Using a low-priority deadlock freedom layer causes throughput-reducing feedback throughout the network.

There is, however, a trade-off. Transferring all misrouted traffic to the high-priority VC3 has an adverse affect on native TC3 traffic, as is evident from the load case 4, where latency increases for approaches 6, 10, and in part 14. This last one, 14, does not show such large increase in latency as the other two, since packets are only switched to the high-priority VC3 when encountering faults in the downward phase, as opposed to both the upward and downward phases in the other two approaches. The optimal solution to this trade-off is apparently to give high priority only to the deadlock freedom channel VC4, without further changing the priorities of packets, represented by approach 4 for which the latency only increases by 50% at the high load case 4.

Let us then focus on the change in latency for traffic sharing links with other traffic directly affected by the fault, but not encountering the faults itself, for traffic classes TC3, TC2, and TC1 in figures 5(a), 5(b), and 5(c), respectively. For TC3 we see that the latency increases in much the same manner as for traffic not affected by the faults, except that the latency increases are generally a bit larger, especially for load case 1. Comparing this to the latency change for the other two traffic classes, we see that there is a marked difference for the highest load, load case 4. For TC2 traffic the effect of the different approaches are much the same as for TC3, while for TC1 traffic the different approaches have little impact on the latency. The reason for the small change in load case 4 latency for TC2 (the bars are not visible behind load case 3) is that its corresponding virtual channels are heavily saturated and latency is thus given by the buffer sizes and path length. Increasing the path length by misrouting around faults will therefore not increase latency any more than the buffer time of the extra path length. The same argument is valid for TC1 traffic which is even more



(a) Affected, but not misrouted, TC3 (b) Affected, but not misrouted, TC2

(c) Affected, but not misrouted, TC1 (d) Misrouted, TC3

Fig. 5. Change in latency from 0-3 faults for 4 loads, all traffic classes

saturated, but there is also an added effect. Even when the deadlock freedom channels are configured with the lowest priority used for these in the simulations, their weight in the arbitration table is still 10 times higher than for VC1, so any misrouting may actually decrease latency by allowing the misrouted traffic of TC1 the same priority as all other traffic misrouted around the fault. This effect will to some degree be present for the other loads as well, but it is not as apparent since VC1 is more saturated.

Finally, we analyse the performance of the TC3 traffic which is directly affected by the fault through having to be misrouted around failures. This is depicted in figure 5(d). On a general note we see that the latency increase for this traffic is significantly larger than for traffic not directly encountering the faults, with the maximum latency increase close to 3000%. Again we see the same classification as previously, with the approaches with high-priority deadlock freedom channels giving the by far best results. Similarely to the other figures, approach 4 yields the lowest increase in latency also for high-priority traffic directly affected by the fault.

Let us now summarise the results. Because of the flow control, giving rerouted traffic high priority has the most beneficial impact on performance. Furthermore, because traffic from all classes will cross the same virtual channel when it is routed in the downward phase, the priority of this channel has the greatest impact on performance. The results also show that maintaining high network efficiency generally works well together with maintaining guarantees for high-priority traffic not directly encountering the faults. Configuring the deadlock freedom channel with a high priority is consequently sufficient to maintain high network utilisation, while at the same time ensuring minimal impact on high-priority traffic in the network.

Admittedly, it is not possible to maintain strict guarantees, but the best approach (4) suffers at most a 100% increase in latency for TC3-traffic that is indirectly affected by the fault at very high loads with three faults. At low load the latency is barely affected, even for the saturated load case 3.

7 Conclusion

Maintaining quality-of-service guarantees in the presence of network faults is difficult when assuming dynamic fault tolerance, as any fault will move traffic around in the network and disrupt the service provided to the jobs. This is especially difficult in Utility Computer Data Centre (UCDC) systems consisting of several virtual servers, each with their own Service Level Agreements. We have shown that it is important to consider traffic priorities when configuring a dynamic rerouting fault tolerance mechanism. Lack of consideration for such properties may in the worst-case lead to severely degraded network performance for high-priority traffic with faults in the network. We have presented a strategy for reprioritising traffic encountering network faults and evaluated a large number of combinations of the possibilities within this strategy. We found that by correctly changing priorities of the packets that encounter network faults, it is to large degree possible to satisfy the same Service Level Agreements as before the fault occurred. However, finding the ultimate configuration is difficult since requirements of the solutions vary greatly between jobs. We have demonstrated that we are able to satisfy a large number of these requirements, and surprisingly most of the requirements could be satisfied by giving misrouted traffic higher priority, even for maintaining high-priority guarantees for high-priority traffic.

References

1. Alfaro, F.J., Sanchez, J.L., Duato, J., Das, C.R.: A strategy to compute the infiniband arbitration tables. In: Proceedings of International Parallel and Distributed Processing Symposium (April 2002)
2. Alfaro, F.J., Sanchez, J.L., Duato, J.: A strategy to manage time sensitive traffic in infiniband. In: Proceedings of Workshop on Communication Architechture for Clusters (CAC) (April 2002)
3. Beecroft, J., Addison, D., Hewson, D., McLaren, M., Roweth, D., Petrini, F., Nieplocha, J.: Qsnetii: Defining high-performance network design. IEEE Micro. 25(4), 34–47 (2005)

4. Chalasani, S., Raghavendra, C.S., Varma, A.: Fault-tolerant routing in MIN based supercomputers. In: Supercomputing 1990: Proceedings of the 1990 conference on Supercomputing, pp. 244–253. IEEE Computer Society Press, Los Alamitos (1990)
5. Myrinet Inc. Myrinet overview (2007), `http://www.myri.com/myrinet/overview/`
6. J-sim (May 2006), `http://www.j-sim.org/`
7. Lee, T.-H., Chou, J.-J.: Some directed graph theorems for testing the dynamic full access property of multistage interconnection networks. In: IEEE TENCON (1993)
8. Leiserson, C.E.: Fat-trees: Universal networks for hardware-efficient supercomputing. IEEE Transactions on Computers C-34(10), 892–901 (1985)
9. Martinez, R., Alfaro, F.J., Sanchez, J.L.: Decoupling the bandwidth and latency bounding for table-based schedulers. In: Proceedings of the 2006 International Conference on Parallel Processing, pp. 155–163 (2006)
10. Petrini, F., Vanneschi, M.: K-ary N-trees: High performance networks for massively parallel architectures. Technical Report TR-95-18, 15 (1995)
11. Sem-Jacobsen, F.O., Lysne, O., Skeie, T.: Combining source routing and dynamic fault tolerance. In: De Souza, A.F., Buyya, R., Meira Jr., W. (eds.) Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Washington, DC, USA, pp. 151–158. IEEE Computer Society, Los Alamitos (2006)
12. Sem-Jacobsen, F.O., Skeie, T., Lysne, O.: A dynamic fault-torlerant routing algorithm for fat-trees. In: International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA, June 27- 30. CSREA Press (2005)
13. Sem-Jacobsen, F.O., Skeie, T., Lysne, O., Duato, J.: Dynamic fault tolerance with misrouting in fat trees. In: Feng, W.c. (ed.) Proceedings of the International Conference on Parallel Processing (ICPP), pp. 33–45. IEEE Computer Society, Los Alamitos (2006)
14. Sem-Jacobsen, F.O., Skeie, T., Lysne, O.: Dynamic fault tolerance in multistage interconnection networks (2008), Research note, Simula, `http://simula.no/research/networks/publications/simula.nd.121`
15. Sem-Jacobsen, F.O., Skeie, T., Lysne, O., Tørudbakken, O., Rongved, E., Johnsen, B.: Siamese-twin: A dynamically fault tolerant fat tree. In: Proceedings of the 19th IPDPS (2005)
16. Sengupta, J., Bansal, P.K.: Fault-tolerant routing in irregular MINs. In: TENCON 1998. 1998 IEEE Region 10 International Conference on Global Connectivity in Energy, Computer, Communication and Control, vol. 2, pp. 638–641 (1998)
17. Sengupta, J., Bansal, P.K.: High speed dynamic fault-tolerance. In: Proceedings of IEEE Region 10 International Conference on Electrical and Electronic Technology, 2001. TENCON, vol. 2, pp. 669–675 (2001)
18. Sharma, N.K.: Fault-tolerance of a MIN using hybrid redundancy. In: Proceedings of the 27th Annual Simulation Symposium, pp. 142–149 (April 1994)
19. Skeie, T.: A fault-tolerant method for wormhole multistage networks. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 1998), pp. 637–644 (1998)

Designing a High-Performance Clustered NAS: A Case Study with pNFS over RDMA on InfiniBand*

Ranjit Noronha, Xiangyong Ouyang, and Dhabaleswar K. Panda

Network Based Computing Lab
Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210
{noronha,ouyangx,panda}@cse.ohio-state.edu

Abstract. Large scale scientific and commercial applications consume and produce petabytes of data. This data needs to be safely stored, cataloged and reproduced with high-performance. The current generation of single headed NAS (Network Attached Storage) based systems such as NFS is not able to provide an acceptable level of performance to these types of demanding applications. Clustered NAS have evolved to meet the storage demands of these demanding applications. However, the performance of these Clustered NAS solutions is limited by the communication protocol being used, usually TCP/IP. In this paper, we propose, design and evaluate a clustered NAS; pNFS over RDMA on InfiniBand. Our results show that for a sequential workload on 8 data servers, the pNFS over RDMA design can achieve a peak aggregate Read throughput of up to 5,029 MB/s, a maximum improvement of 188% over the TCP/IP transport and a Write throughput of 1,872 MB/s; a maximum improvement of 150% over the corresponding TCP/IP transport throughput. Evaluations with other type of workloads and traces show an improvement in performance of up to 27%. Finally, our design of pNFS over RDMA improves the performance of BTIO relative to the Lustre file system.

1 Introduction

The explosive growth in multimedia, Internet and other content have caused a dramatic increase in the volume of media that needs to stored, cataloged and accessed efficiently. In addition, high-performance applications on large supercomputers process and create petabytes of application and checkpoint data. Modern single-headed nodes with a large number of disks (single headed Network Attached Storage (NAS)) may not have the adequate capacity to store this data. Also, the single head or single server may potentially become a bottleneck with accesses from a large number of clients. Also, a failure of the node or the disk may lead to a loss of data.

To deal with several of these problems, clustered NAS solutions have evolved. Clustered NAS solutions attempt to store the data across a number of storage servers. This

* This research is supported in part by Department of Energy's grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation's grants #CNS-0403342 and #CCF-0702675; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Cisco Systems, Intel, Mellanox, QLogic, Sun Microsystems and Linux Networks; and equipment donations from Advanced Clustering, AMD, Appro, Intel, Mellanox, Microway, Qlogic and Sun Microsystems.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 465–477, 2008.

has a number of benefits. First, we are no longer limited to the capacity of a single node. Second, depending on the way data is stripped across the nodes, with accesses from a large number of clients, the load will be more evenly distributed across the servers. Third, for large files, this architecture has the advantages of multiple streams of data from different nodes for better aggregate bandwidth for larger file sizes. Finally, clustered NAS allows data to be stored redundantly across a number of different nodes [1], reducing the likelyhood of data loss.

Even though clustered NAS provides several benefits in term of capacity, enhanced load capacity, better aggregate throughput and better fault-tolerance, they bring with them their own set of unique problems. First, since the data-servers have now been de-coupled, any given stream of data will require multiple network, usually TCP/IP, connections from the clients to the data servers and metadata servers. TCP/IP connections have been shown to have considerable overhead, mainly in terms of copying costs, fragmentation and reassembly, reliability and congestion control. In addition, with multiple streams of incoming data from multiple data-servers, TCP/IP connections have been shown to suffer from the problem of incast [2], which severely reduces the throughput. Second, TCP/IP with multiple copies and considerable overhead is unable to take advantage of the high-performance networks like InfiniBand and 10GigE. Third, with a single headed NAS, there is only a single point of failure, making it easier to protect the data on the NAS. However, with a clustered NAS, we have multiple data servers, with multiple failure points.

Modern high-performance networks such as InfiniBand provide low-latency and high-bandwidth communication. For example, the current generation ConnectX NIC from Mellanox has a 4 byte message latency of around 1μs and a bi-directional bandwidth of up to 4 GB/s for large messages. Applications can also deploy mechanisms like Remote Direct Memory Access (RDMA) for zero-copy low-overhead communication. RDMA operations allow two appropriately authorized peers to read and write data directly from each other's address space. RDMA requires minimal CPU involvement on the local end, and no CPU involvement on the remote end. Designing the stack with RDMA may eliminate the copy overhead inherent in the TCP and UDP stacks and reduce CPU utilization. As a result, a high-performance RDMA enabled network like InfiniBand might potentially reduce the overhead of TCP/IP connections in clustered NAS.

In our earlier work, we designed a Network File System (NFS) (which is a single headed NAS) with RDMA operations in InfiniBand [3] for NFSv3 and NFSv4. In this paper, we propose, design and evaluate a clustered Network Attached Storage (NAS). This clustered NAS is based on parallel NFS (pNFS) with RDMA operations in InfiniBand. While other parallel and clustered file systems such as Lustre [1] exist, we choose pNFS since NFS is widely deployed and used. In this paper, we make the following contributions:

- An in-depth discussion of the tradeoffs in designing a high-performance pNFS with an RPC/RDMA transport.
- An understanding of the issues with sessions that provides exactly once semantics in the face of network faults and the trade-offs in designing pNFS with sessions over RDMA.
- A comprehensive performance evaluation with micro-benchmarks and applications of a RDMA enabled pNFS design.

Our evaluations show that by enabling pNFS with an RDMA transport, we can decrease the latency for small operations by up to 65% in some cases. Also, pNFS enabled with RDMA allows us to achieve a peak IOzone Write and Read aggregate throughput of 1,872 MB/s and 5,029 MB/s, respectively using a sequential trace with 8 data servers. The RDMA enabled Write and Read aggregate throughput is 150% and 188% better than the corresponding throughput with a TCP/IP transport. Also, evaluation with a Zipf trace distribution allows us to achieve a maximum improvement of up to 27% when switching transports from RDMA to TCP/IP. Finally, application evaluation with BTIO shows that the RDMA enabled transport with pNFS performs better than with a TCP/IP transport by up to 8.8% and better than Lustre by up to 22%.

The rest of the paper is presented as follows. Then, Section 2 looks at the parallel NFS and sessions extensions to NFSv4.1. Following that, Section 3 looks at the design considerations for pNFS over RDMA. After that, Section 4 evaluates the performance of the design. We present related work in Section 5. Finally, Section 6 discusses conclusions and future work.

2 NFSv4.1: Parallel NFS (pNFS) and Sessions

In this section, we discuss pNFS and sessions, which are defined by the NFSv4.1 semantics.

Parallel NFS (pNFS): The NFSv4.1 [4] standard defines two main components; namely parallel NFS (pNFS) and sessions. The focus of pNFS is to make an NFSv4.1 client a front-end for clustered NAS or parallel file-system. The pNFS architecture is shown in Figure 1. The NFSv4.1 client can communicate with any parallel file using the *Layout* and *I/O driver* in concert with communications with the NFSv4.1 server. The NFSv4.1 server has multiple roles. It acts as a metadata server (MDS) for the parallel/cluster file system. It sends information to the client on how to access the back-end cluster file system. This information takes the form of *GETDEVICEINFO*, which returns information about a specific data-server in the cluster file system, usually an IP address and port number that is stored by the client layout driver. The NFSv4.1 server is also responsible for communicating with the data servers for file creation and deletion. The NFSv4.1 server may either directly communicate with the data servers, or it may communicate with a metadata server, that is responsible for talking to and controlling the data servers in the parallel file system. The pNFS client uses the file layout and I/O driver for communicating with the data servers. The layout driver is responsible for translating READ and WRITE requests from the upper layer into the corresponding protocol that the back-end parallel/cluster file system uses; namely object, block and file. This is achieved through the additional NFS procedures *GETFILELAYOUT* (how the file is distributed across the data servers), *RETURNFILELAYOUT* (after a file is closed), *LAYOUTCOMMIT* (commit changes to file layout at the metadata server, after writes have been committed to data servers). Examples of pNFS designs are discussed further in the technical report [5].

NFSv4.1 and sessions: Sessions are aimed at making the NFSv4 non-idempotent requests resilient to network level faults. Traditionally, non-idempotent requests are taken care of through the Duplicate Request Cache (DRC) at the server. The DRC has a limited number of entries, and these entries are shared among all the clients. So, eventually some entries will be evicted from the cache. In the face of network-level partitions, duplicate requests that arrive that have been evicted from the DRC, will be re-executed. Sessions solve this problem by requiring each connection to be alloted a fixed number of RPC slots in the DRC. The client is only allowed to issue requests up to the number of slots in the connection. Because of this reservation policy, duplicate requests from the client to the server in the face of network-level partitions will not be re-executed. We will consider design issues with sessions and RPC/RDMA in the following section.

RPC/RDMA for NFS: The existing RPC/RDMA design for Linux and OpenSolaris is based on the Read-Write design [3]. It consists of two protocols; namely the inline protocol for small requests and the bulk data transfer protocol for large operations. The inline protocol on Linux is enabled through the use of a set of persistent buffers; (32 buffers of 1K each for Send and 32 buffers of 1K each for receives on Linux). RPC Requests are sent using the persistent inline buffers. RPC replies are also received using the persistent inline buffers. The responses for some NFS procedures such as READ and READDIR might be quite large. These responses may be sent to the user via the bulk-data transfer protocol, which uses RDMA Write to send large responses from the server to the clients without a copy and RDMA Reads to pull data in from the client for procedures such as Write. The design trade-offs for RPC/RDMA are discussed further in [3].

3 Design Considerations for pNFS over RDMA

In this section, we examine the considerations for a high-performance design of pNFS over RDMA. First, we look at the detailed architecture of pNFS with a file layout driver.

3.1 Design of pNFS Using a File Layout

As discussed in Section 2, pNFS can potentially use an object, block or file based model. In this paper, we use the file-based model for designing the pNFS architecture. We now discuss the high-level design of the pNFS architecture.

pNFS Architecture: The detailed architecture is shown in Figure 2. The NFSv4.1 clients use a file layout driver which is responsible for communicating with the NFSv4 servers, that act as the data-servers. At the NFSv4.1 server, the sPNFS daemon runs in user-space. The sPNFS daemon communicates with the NFSv4.1 server in the kernel via the RPC PipeFS. The RPC PipeFS is essentially a wait queue in the kernel. The NFSv4.1 server enqueues requests from the clients via the control path, and these requests are then pushed to the sPNFS daemon via an upcall. The sPNFS daemon then processes each of these requests and makes a downcall into the kernel with the appropriate data/response for the requests. The NFSv4.1 requests which are sent up to the sPNFS daemon for processing include the NFSv4.1 procedures *GETFILELAYOUT*, *RETURNFILELAYOUT*, *LAYOUTCOMMIT* and *GETDEVICEINFO*. These procedures are discussed in Section 2. In-order to work on the processing of the requests, the sPNFS

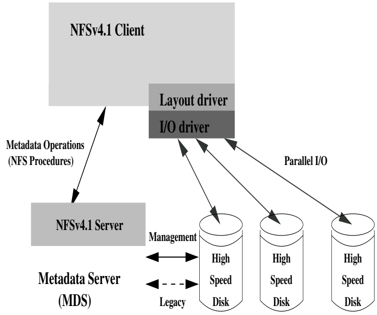


Fig. 1. pNFS high-level architecture

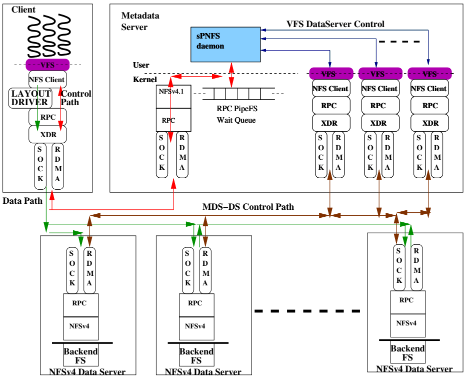


Fig. 2. pNFS detailed design

daemon mounts an NFSv3 directory from each of the data-servers. For example, when a file layout is requested (*GETFILELAYOUT*), the sPNFS daemon may need to create the file on each of the data servers or open the file through the *VFS DataServer Control Path*.

sPNFS file creation: To create a file, the sPNFS daemon will open the file on the mount of each of the data servers in create mode. It will then do a *stat* to make sure that the file actually got created or exists. It will then close the file (the file handle is static). This traffic will propagate via RPC calls through the *MDS-DS control path*. Finally, it will return the set of open file descriptor to the NFSv4.1 server as part of the response to the upcall. The NFSv4.1 server will then reply to the NFSv4.1 client with the file layout. The client will then use the layout received (through the file layout driver) to communicate with the NFSv4 data servers using the *Data Paths*.

sPNFS file deletion: File deletes are initiated by the NFS REMOVE procedure. The REMOVE procedure is sent up to the sPNFS daemon through RPC PipeFS. The process of deleting a file is opposite to that of creation. The sPNFS daemon will try to delete each of the file from the mount points. Once this is achieved, sPNFS will send a message to the NFS kernel thread about this.

3.2 RPC Connections from Clients to MDS (*Control Path*)

The RPC connections from the clients to the MDS may be through either RDMA or TCP/IP. A majority of the communication from the clients to the metadata server is expected to be small operations or metadata intensive workloads. As a result, these workloads may potentially benefit from the lower latency of RDMA. However, since NFS and RPC are in the kernel, there is the cost of a context switch from user-space to kernel-space, in addition to the copying costs with the NFS and RPC stacks. Depending on factors such as the CPU speed and memory bus-bandwidth, these costs might dominate. Correspondingly, the lower latency of RDMA might not provide much of a benefit in these cases. Another important factor that needs to be considered is the memory utilization and scalability of the MDS. The MDS is required to maintain RDMA enabled

RPC connections with all the clients. Each of these connection holds 32 1K send buffers and 32 2K receive buffers. These buffers are not shared across all the connections. With a very large number of client connections using RPC over RDMA, the MDS server might run out of buffers that might be appropriately utilized. In these cases, using RPC over TCP might be more appropriate for the majority of clients, though the high copying cost associated with TCP/IP connections needs to be considered. If an RDMA enabled RPC transport can provide adequate benefit for small operations, it might be appropriate to use a few connections with RDMA for some clients that communicate frequently with the MDS and a TCP enabled RPC transport for the remaining connections. A final factor that needs to be considered is the disconnect time for a RDMA enabled RPC transport. RDMA enabled RPC connections are disconnected after 2 minutes idle time. Reestablishing a RDMA enabled RPC connection is a very expensive operation because of the high-overhead of registering memory and reestablishing the eager protocol [3]. In comparison, RPC over TCP does not have such high-latencies for reestablishing the connections.

3.3 RPC Connections from MDS to DS (*MDS-DS Control Path*)

It might be potentially possible to use RPC over RDMA or RPC over TCP connections between the MDS and DSes. The *MDS-DS control path* allows the MDS to control the NFSv4 data-servers. This control is in the form of file creations and deletions. There are a number of factors that affect the choice of a RPC enabled with RDMA or TCP connection from the metadata server to the data servers. As discussed earlier, the sPNFS daemon is multi-threaded. As a result, there are expected to be a large number of requests in flight, in parallel. So, the lower potential latency of RPC with RDMA is likely to provide a benefit in completion of these requests. Also, the fixed number of buffers per connection is expected to provide a better flow-control mechanism for a large number of outstanding parallel requests. Finally, the number of data servers is relatively small in comparison to the number of clients. As a result, the *MDS-DS control path* is not likely to be severely affected by the buffer scalability issue that may potentially affect the *Control Path*.

3.4 RPC Connections from Clients to Data Servers (*Data Paths*)

The expected traffic patterns from the client to the data servers is expected to consist of small, large and medium size traffic. Since 32K is the maximum payload for the cached I/O case, this is likely to be the most common transfer over the network, depending on the stripe size of the file at the data servers. We also need to consider the case of buffer scalability. Since data-servers are expected to have connections from a large number of clients, and since each connection will have persistent buffers, this might cause a memory scalability issue. However, clients do not connect to a particular data server unless the data server is in the list of DSes returned in the file layout. As a result, not all clients will be connected to all data servers at any given time. Depending on the load on the back-end file system, using an RPC over RDMA connection from the data-servers to the client might not cause a large amount of overhead at the data-servers. Also, quiescent clients will be disconnected from the data-servers, further reducing the overhead. Since an RPC transport enabled with RDMA has been shown to provide considerable benefits via-viz large transfers, it might be beneficial to use RPC over RDMA between the clients and data-servers.

3.5 Sessions Design with RDMA

As discussed earlier, sessions provides exactly once semantics for all NFS procedures in the wake of network-level faults. To do this, sessions provide dedicated slots of buffers to each connection between the client and the servers. The client may only send requests up to a maximum number of slots per session. In order to design sessions with a RDMA enabled RPC transport, we associate the inline buffers in each connection with the minimum number of slots required from the connection. If the number of slots requested is lower than the number available, and the caller cannot accept a lower number, the session create request will fail. The disadvantages of the sessions design with RDMA is that advanced features of the InfiniBand network such as the Shared Receive Queue (SRQ) cannot be used. SRQ enhances the buffer scalability by having the buffers shared across all the InfiniBand connections. When the number of buffers falls below a certain watermark, an interrupt may be generated to post more buffers. Since sessions require that slots be guaranteed per connection, SRQ cannot be used.

4 Performance Evaluation

In this section, we evaluate the performance of pNFS designed with an RPC over RDMA transport First, We discuss the experimental setup in Section 4.1. Following that, in Sections 4.2, 4.3 and 4.4, we look at the relative performance advantages of using an RDMA enabled RPC transport over a TCP/IP transport in different configurations involving the metadata server (MDS), Data Server (DS) and Client. Since sessions only requires reservation of RDMA inline buffers, we do not evaluate the sessions portion of the design.

4.1 Experimental Setup

To evaluate the performance of the RPC over RDMA enabled pNFS design (pNFS/RDMA), we used a 32-node cluster. Each node in the cluster is equipped with a dual Intel Xeon 3.6 GHz CPU and 2GB main memory. For InfiniBand communication, each node uses a Mellanox Double Data Rate (DDR) HCA. The nodes are equipped with SATA drivers, which are used to mount the backend ext3 filesystem. A memory based filesystem ramfs is also used for some experiments. The pNFS with sockets uses IP over InfiniBand (IPoIB) and we refer to this transport as pNFS/IPoIB. We use pNFS/IPoIB and pNFS/TCP interchangeably. All experiments using IPoIB are based on Reliable Connection mode (IPoIB-RC) and an MTU of 64KB, unless otherwise noted. We explicitly use pNFS/IPoIB-UD to explicitly mean an unreliable datagram mode of transport. IPoIB-UD uses a 2K MTU size.

4.2 Impact of RPC/RDMA on Performance from the Client to the Metadata Server

The clients communicate with the MDS using either NFSv4 or NFSv4.1 procedures. As Section 2 mentions, the vast majority of NFSv4.1 requests from the clients to the MDS are expected to be procedures such as *GETDEVICEINFO*, *GETDEVICELIST*, *GETFILELAYOUT* and *RETURNFILELAYOUT*. These small procedures will potentially carry small and medium size payloads. For example, *GETFILELAYOUT* returns a list of file handles, which is only a small amount of payload. A file handle can be encoded with no more than 16 bytes of information (although a native file handle size may vary depending on platforms). One of the largest deployments of a parallel file system Lustre [1] in recent times is the TACC [6] cluster with 8,000 nodes containing 64,000 cores, serviced by a bank of 1,000 data server nodes. With 1,000 data server nodes and the assumption that a file is stripped across all the data server nodes, the payload from *GETFILELAYOUT* will only be 16K. Also, some of these operations such as *GETDEVICEINFO* are only executed at mount time and are not in the critical path. On the other hand, operations such as *CREATED*, *GETFILELAYOUT*, *RETURNFILELAYOUT* are executed every time a file is created, opened and closed. With a workload consisting of a large number of such operations (metadata intensive workloads) RPC/RDMA is likely to provide some benefit. Also, *LAYOUTCOMMIT* is executed once a WRITE operation completes and is likely to be in the critical path for workloads dominated by write operations. To understand the relative performance of small operations when switching transports from RPC/TCP to RPC/RDMA, we measured the latency of issuing a *GETFILELAYOUT* (at the RPC layer) from the client to the MDS and the time required for it to complete, averaged over 1024 times, while the payload from the MDS to the client was varied from 1 to 32K bytes. A 32K message can contain the information for more than 2,000 file handles and might be considered large for contemporary, high-performance parallel file system deployments. The measured latency is shown in Figure 3. As shown in Figure 3, the latency with a 1 byte payload is $68\mu s$ with RPC/RDMA and $71\mu s$ with RPC/TCP. The relatively low improvement in performance is because the high access latency of the disk which is a dominant portion of the latency. With larger access, the disk blocks are prefetched because of sequential access and the performance improvement from using RPC/RDMA is increased by up to 65%. The performance benefit of the RPC/RDMA connection from the client to the MDS is taken in the context of the inline buffers, which need to be statically allocated per-client at the MDS. With an increasing number of clients, the RPC/RDMA connections may consume considerable memory resources. Since the MDS is likely to be the target of a mainly metadata intensive workload, it becomes imperative to maintain a large number of inline buffers in order to guarantee a high throughput performance.

4.3 RPC/RDMA Versus RPC/TCP on Metadata Server to Data Server

The connection from the MDS to the DSes may also consist of RPC/RDMA. The sPNFS daemon controls the DSes by mounting the exported directories from the data servers. The sPNFS daemon creates, open and deletes files in the exported directories. These calls are translated through the VFS layer to RPC/RDMA calls. Thus the scalability of these calls is directly impacted by the time required by the RPC operations to complete. To gain insight into the relative scalability of the RPC/RDMA and RPC/TCP transports, we measured the performance of create portion of the sPNFS daemons operation. In this multi-process benchmark, each process is synchronized in the start phase by a barrier. After being released from the barrier, each process performs a stat operation on the target file to check its state, then opens this file in creation mode. These two operations are followed by a chmod to set the mode of this file, and a close operation to close this file. The close operation is a portion of the process to open a pNFS file, and it is included to avoid running out of open file handles, a limited operating system resource. Each process performs each of these operations on every one of the DS mounts. The time required for 1024 of these operations is measured and averaged out. This test is performed for a RPC/RDMA and RPC/TCP transport from the MDS to the DSes. These numbers are shown in Figures 4 and 5. In Figures 4 and 5, we observe the following trends. RPC/RDMA performs worse than RPC/TCP (indicated as IPoIB) for 1 process. Note that in this case, we are measuring the time at the VFS level, whereas in Section 4.2, we are measuring the time at the RPC layer. In the current scenario, the IPoIB-RC driver uses a ring of 128 receive buffer of size 64K and 64 send buffers. On the other hand, RPC/RDMA uses 32-buffers of 1K. As a result, with an increasing number of data servers, and 1 process, more create and stat operations can be issued in parallel with IPoIB-RC than with RPC/RDMA (we issue 1,024 create operations and 1,024 stat operations for a total of 2,048 operations). However, with IPoIB-RC all 128 receive buffers are shared across all the connections using SRQ. With RPC/RDMA, each connection from an MDS to a DS is allocated a set of 32-buffers. As a result, when the number of connections increases, RPC/RDMA has a dedicated set of buffers in which to receive messages, while IPoIB has a fixed number of buffers, and this might result in dropped messages with IPoIB. Also in RPC/IPoIB, there are up to 5 copies from the application to the IP-level. With RPC/RDMA, there are up to 3 copies from the application down to the RPC/RDMA layer. With an increasing number of processes, the larger number of copies in the case of IPoIB begins to dominate and IPoIB performs worse than RDMA. The copying cost with IPoIB and 1 client does not totally consume the CPU and so is not the dominant factor. As a result, with 1 process, RPC/TCP is able to perform better than RPC/RDMA. At 2 processes per-node, RPC/RDMA and RPC/TCP

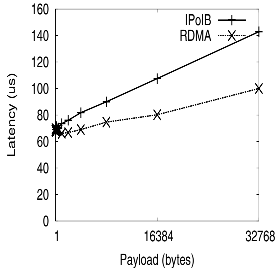


Fig. 3. Small Operations

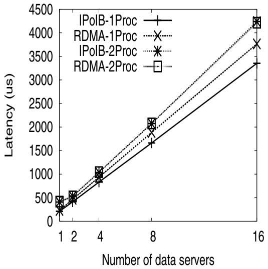


Fig. 4. Latency for 1 and 2 processes

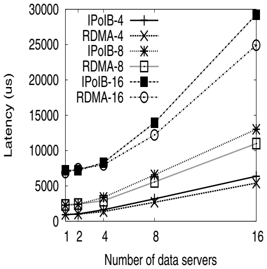


Fig. 5. Latency for 4,8 and 16 processes

perform comparably with an increasing number of data servers. At 4 processes/node and above with RPC/RDMA, the time required to perform the create operations is lower than RPC/TCP. At 16 processes at the MDS, the improvement with 16 DSes there is a maximum decrease in latency of 15%. The trends we have observed indicate that RPC/RDMA will perform better than RPC/TCP with a larger volume of operations. We have conducted a test with 32 client threads with both RPC/RDMA and RPC/TCP. RPC/RDMA exhibits similar degree of improvement over RPC/TCP. Analysis shows that the SRQ used in IPoIB plays a role in the performance reduction with RPC/TCP and is discussed further in the technical report [5].

4.4 RPC/RDMA Versus RPC/TCP from Clients to DS

We measure the relative performance impact of changing the transport from RPC/RDMA to RPC/TCP from the client to the data servers. To measure the performance impact, we use three different benchmarks: sequential throughput with IOzone, throughput of a Zipf trace and a parallel application BTIO.

Sequential Throughput. We use IOzone [7] in cluster mode to measure the performance of a sequential workload modeling the throughput from the client to the DSes. 8 nodes act as data servers, 8 nodes act as clients, and 1 node is designated ad the metadata server. Each client node hosts one IOzone process. The benchmark is run on both the IPoIB Reliable Connection mode (IPoIB-RC) and IPoIB Unreliable Datagram mode (IPoIB-UD) to compare against RPC/RDMA. The IOzone record size is kept at 32KB, the default cached I/O maximum size and the total file size per client used is 512MB. The Write and Read throughput while varying the number of data servers and clients (aggregate throughput) is shown in Figures 6 and 7, respectively.

For Write, RPC/RDMA begins to outperform RPC/TCP as the number of data server is increased beyond two. At 8 data servers and 8 clients, RPC/RDMA reaches its peak write throughput of 1,872 MB/s, which is 22% higher than IPoIB-RC and 150% higher than IPoIB-UD. For Read, there is an improvement in performance for all cases. Using RPC/RDMA achieves a peak read throughput of 5,029 MB/s at 8 clients and 8 data servers, which outperforms IPoIB-RC by 89% and IPoIB-UD by 188%.

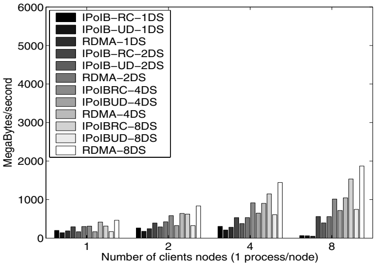


Fig. 6. IOzone Throughput (Write)

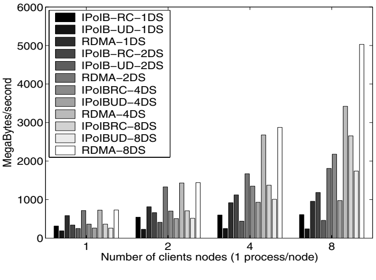


Fig. 7. IOzone Throughput (Read)

Throughput with a Zip Trace. Zipf's law, named after the Harvard linguistic professor George Kingsley Zipf (1902-1950), is the observation that frequency of occurrence of some event (P), as a function of the rank (i) when the rank is determined by the above frequency of occurrence, is a power-law function $P_i \sim 1/i^a$ with the exponent α close to unity. Zipf distributions have been shown to occur in a variety of different environments such as word distributions in documents, web-page access patterns and file and block distributions in storage sub-systems [8].

We modified IOzone to issue write and read requests, where the size and location of the Read or Write request follows a Zipf distribution with an α=0.9. We used IOzone to measure the throughput of the trace on a single node with one thread, issuing requests where the location and I/O size of the issued request follows a Zipf distribution. We used a 512MB file size on both an ext3 as well as a *ramfs* backend file system. The *ramfs* file system streams data from memory and serves as an upper bound on the performance improvement we can expect with pNFS/RDMA. We compare pNFS/RDMA with pNFS/IPoIB-RC while varying the number of data servers. The results for Write are shown in Figure 8, while the results for Read are shown in Figure 9. We observe that the RPC transport used does not have a large impact on performance for Writes. Disk Filesystem Write performance is generally sensitive to the performance of the backend storage subsystem. The large majority of disks exhibit poor random Write performance. Also, depending on the organization of the in-memory file system, *ramfs* based systems have also been shown to exhibit poor performance for random Write operations. Correspondingly, for the Zipf based Write distribution, we see a very poor throughput of around 500 MB/s for both pNFS/RDMA and pNFS/IPoIB-RC. Since the CPU is not fully utilized for TCP/IP when the throughput is lower, we expect little improvement from pNFS/RDMA over pNFS/IPoIB-RC. On the other hand, the IOzone Read throughput is impacted by the underlying RPC transport. With a *ramfs* based file system, we see an improvement of 22% from pNFS/IPoIB-RC to pNFS/RDMA with 1 data server. The improvement in throughput from pNFS/IPoIB-RC to pNFS/RDMA increases to 27% at 8 data servers. We are also able to achieve a peak throughput of 2073 MB/s with the Zipf trace at 8 data servers. Since, the Zipf trace has an element of randomness, a portion of the Read data is cached at the client. As a result we see some amount of cache effect in addition to network-level transfers, which reduces the potential performance improvement with pNFS/RDMA. Techniques for improving the performance of pNFS/RDMA for a Zipf workload are discussed further in the technical report [5].

Performance with a Scientific Kernel NAS BTIO. The NAS Parallel Benchmarks (NPB) suite discussed further in the technical report [5], is used to measure the performance of Computational Fluid Dynamic (CFD) codes on a parallel machine. One of the benchmarks BT measures the performance of block-triangulation equations in parallel. In addition to the computational phase of BT, BTIO adds additional code for checkpointing and verifying the data from the computation. We use the Full MPI I/O mode in which MPI collective calls are used to aggregate Read and Write operations. We run BTIO with a class A size (that uses a 64x64x64 array) over pNFS/RDMA, pNFS/IPoIB and Lustre. In this configuration, BTIO generates a file of size 400 MB. The results with an ext3 back-end file system at eight data servers are shown in Figure 10.

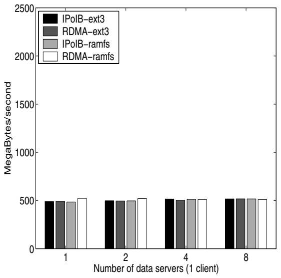


Fig. 8. Zipf trace (Write)

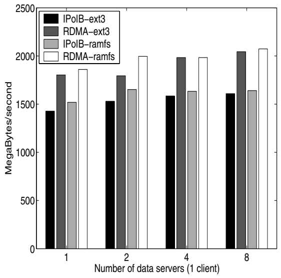


Fig. 9. Zipf trace (Read)

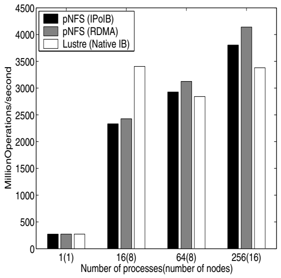


Fig. 10. BTIO with ext3

In this experiment, we measured the performance of BTIO with eight data servers. We used one, 16, 64 and 256 processes (BTIO requires a square number of processes). For the 16 process case, we use eight client nodes (two processes/node). For the 64 and 256 process case, we also use eight and sixteen client nodes respectively (eight processes per node and 16 processes/node respectively). We observe the following trends. First, pNFS/IPoIB-RC and pNFS/RDMA perform comparably at one process on one node. As the number of processes increases, pNFS/RDMA begins to perform better than pNFS/IPoIB-RC. At 16 processes (two processes/node), BTIO over a pNFS/RDMA transport performs up to 4% better than over a pNFS/IPoIB-RC transport. At 64 processes (eight processes/node) and 256 processes (16 processes/node), this increases to approximately 7% and 8.8%, respectively. The better bandwidth of the underlying transport helps improve MPI collective I/O performance and is discussed further in the technical report [5]. Finally, we also compared with Lustre using a native InfiniBand transport and eight data servers. pNFS/RDMA outperforms Lustre by up to 22% at 256 processes.

5 Related Work

There are a large number of single headed NAS, clustered NAS storage system and parallel file-systems. Network File System (NFS) is one of the most popular single headed NAS systems. RPC over RDMA transport for NFS exists on both OpenSolaris and Linux [3]. In our work, we design an RPC over RDMA transport for parallel NFS. Lustre [1] is another popular parallel file system. It also allows access to native InfiniBand through the IB Network Access Layer (NAL). The native InfiniBand NAL uses RDMA operations. Our work differs from the IB NAL of Lustre in that we design RPC directly over RDMA, whereas Lustre uses RPC over portals, which in turn calls the NAL functionality.

6 Conclusions and Future Work

In this paper, we propose, design and evaluate a high-performance clustered NAS. The clustered NAS uses parallel NFS (pNFS) with an RDMA enabled transport. We consider a number of design considerations and trade-offs, in particular, buffer management at the client, DS and MDS, scalability of the connections with increasing number of clients and data servers. We also look at how an RDMA transport may be designed with sessions which gives us exactly once semantics. Our evaluations show that enabling pNFS with a RDMA transport, we can decrease the latency for small operations by up to 65% in some cases. Also, pNFS enabled with RDMA allows us to achieve a peak IOzone Write and Read aggregate throughput of 1,800+ MB/s (150% better than TCP/IP) and 5,000+ MB/s (188% improvement over TCP/IP) respectively, using a sequential trace and 8 data servers. Also, evaluation with a Zipf trace distribution allows us to achieve a maximum improvement of up to 27% when switching transports from RDMA to TCP/IP. Finally, application evaluation with BTIO shows that the RDMA enabled transport with pNFS performs better than a transport with TCP/IP by up to 8.8% and better than Lustre by up to 22%.

As part of future work, we would like to explore how to design a fault tolerant pNFS enabled with RDMA. pNFS allows us to use multi-pathing to enable redundant dataservers. We would also like to explore how the shared receive queue (SRQ) optimization may be used with an RDMA enabled RPC transport that uses sessions. Sessions require the reservation of slots or RDMA eager buffers per RPC connection. Dedicating a fixed number of buffers might have an impact on the scalability of larger systems deployed with pNFS. Finally, we would like to evaluate the scalability of our RDMA enabled pNFS design.

References

1. Zahir, R.: Lustre Storage Networking Transport Layer, `http://www.lustre.org/docs.html`
2. Phanishayee, A., Krevat, E., Vasudevan, V., Andersen, D.G., Ganger, G.R., Gibson, G.A., Seshan, S.: Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. In: Proc. USENIX Conference on File and Storage Technologies, San Jose, CA (Feburary 2008)
3. Noronha, R., Chai, L., Talpey, T., Panda, D.: Designing NFS with RDMA on InfiniBand for Security, Performance and Scalability. In: International Conference on Parallel Processing, Xian, China (2007)
4. Shepler, S., Eisler, M., Noveck, D.: NFS Version 4 Minor Version 1., `http://tools.ietf.org/html/draft-ietf-nfsv4-minorversion1-19`
5. Noronha, R., Ouyang, X., Panda, D.K.: Designing a High-Performance Clustered NAS: A Case Study With pNFS over RDMA on InfiniBand. Technical Report OSU-CISRC-5/08-TR28, Department of Computer Science and Engineering, The Ohio State University (2008)
6. Ranger: Sun Constellation Linux Cluster, `http://www.tacc.utexas.edu/resources/hpcsystems/`
7. IOzone Filesystem Benchmark, `http://www.iozone.org`
8. Batsakis, A., Burns, R., Kanevsky, A., Lentini, J., Talpey, T.: AWOL: An Adaptive Write Optimizations Layer. In: FAST 2008: Proceedings of the 6th USENIX Conference on File and Storage Technologies, Berkeley, CA, USA, pp. 1–14. USENIX Association (2008)

Sockets Direct Protocol for Hybrid Network Stacks: A Case Study with iWARP over 10G Ethernet*

Pavan Balaji[1], Sitha Bhagvat[2], Rajeev Thakur[1], and Dhabaleswar K. Panda[3]

[1] Math. and Comp. Sci., Argonne Natl. Lab
{balaji,thakur}@mcs.anl.gov
[2] Scalable Systems Group, Dell Inc.
sitha_bhagvat@dell.com
[3] Computer Science and Engg., Ohio State University
panda@cse.ohio-state.edu

Abstract. As high-end computing systems continue to grow, the need for advanced networking capabilities, such as hot-spot avoidance and fault tolerance, is becoming important. While the traditional approach of utilizing intelligent network hardware has worked well to achieve high performance, adding more and more features makes the hardware complex and expensive. Consequently, protocol stacks such as iWARP and MX for 10-Gigabit Ethernet and QLogic InfiniBand, utilize hybrid hardware-software designs that take advantage of the processing power of multi-core processors together with network hardware accelerators. However, upper-layer stacks on these networks, such as the Sockets Direct Protocol (SDP), have not kept pace with such shift in paradigm, and have continued to assume complete hardware offload, leading to redundant features and performance loss. In this paper, we propose an enhanced design for SDP that allows network stacks to specify components implemented in hardware and software, and uses this information to optimize its execution.

1 Introduction

As high-end computing (HEC) systems continue to increase rapidly in size, their communication subsystems must scale as well. For large-scale systems, in addition to raw performance, advanced communication features such as capability to avoid hot-spot congestion [29,33] and hardware faults [15] are also becoming increasingly important. While the traditional approach of utilizing intelligent hardware support on the network adapters (e.g., Mellanox InfiniBand [2], Myrinet 2000 [14], Quadrics [28], hardware iWARP [19,23]) has worked well to

* This work was supported in part by the National Science Foundation Grant #0702182 and the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 478–490, 2008.

achieve high performance, adding more and more features makes the hardware complex, error prone, and expensive.

At the same time, there have been prominent advances in processor technology, especially powered by the advent of multi-core architectures [5,25]. Thus, to take advantage of these two trends, several network stacks (e.g., QLogic InfiniBand [30], Myri-10G [27], software iWARP [8]) have started to utilize hybrid hardware-software stack designs (known as *hybrid network stacks*). These hybrid network stacks take advantage of the processing power of multi-core processors together with network hardware accelerators to achieve high performance while providing the flexibility to add most communication features relevant to modern HEC systems.

However, several upper-layer stacks on top of these networks have not been able to keep pace with such shift in paradigm of network communication stacks. For example, existing implementations of high-performance sockets on high-speed networks, such as the Sockets Direct Protocol (SDP) [10] over InfiniBand (IB) [24] and 10-Gigabit Ethernet (10GE) iWARP [31], continue to assume complete hardware offload. Consequently, they perform various tasks, such as data buffering to optimize small message communication and message-level flow-control that allow them to achieve high performance on hardware-offloaded network stacks but are redundant on hybrid network stacks and can add significant performance overheads.

In this paper, we perform a case study with SDP on top of a hybrid hardware-software iWARP design over 10GE, and study the drawbacks of its existing implementation. We also propose an enhanced design for SDP that allows network stacks to specify what components are implemented in hardware and what are implemented in software, and uses this information to avoid redundancy in the overall stack. We experimentally compare our proposed approach with the traditional design of SDP using both micro-benchmarks as well as two real applications (virtual microscope [17] and iso-surface oil-reservoir data visualization [13]) built on top of the DataCutter library [12]. Our results demonstrate that the proposed approach can outperform the traditional approach by nearly 20% in micro-benchmarks and about 5% in real applications.

2 Background

In this section, we present a brief overview of SDP and iWARP implementations.

2.1 Overview of SDP

SDP is a byte-stream transport protocol that closely mimics TCP sockets' stream semantics. It is an industry-standard specification for IB and iWARP that utilizes advanced capabilities provided by the network stacks to achieve high performance without requiring modifications to existing sockets-based applications. SDP is layered on top of IB or iWARP's message-oriented transfer model. The mapping of the byte-stream protocol to the underlying message-oriented semantics was designed to transfer application data by one of two methods: through intermediate private buffers (using buffer copy) or directly between application buffers (zero copy).

Zero-copy Approach: Hardware-offloaded protocol stacks allow zero-copy communication of application data. However, such communication comes with several restrictions. For instance, communication buffers have to be *registered*: (i) they need to be pinned so that their physical memory pages cannot be swapped out and (ii) the virtual-to-physical address translation must be provided to the communication stack to potentially be cached on the network adapter. Also, to perform zero-copy communication in SDP, the sender and the receiver have to synchronize on the source and destination buffers, which adds overhead. Thus, while zero-copy communication avoids memory copies, it adds other overheads. Accordingly, SDP uses it only for transferring large messages.

Buffer-copy Approach: Due to the overheads of zero-copy communication, SDP utilizes a buffer-copy approach for small messages. In this approach, it pre-registers private buffers at connection-establishment time. On a send, the data is copied into the registered private buffers, communication carried out from and to these buffers, and finally the data copied out to the destination application buffer on the receiver side. However, the buffer-copy approach also comes with two disadvantages. First, data that needs to be communicated has to be copied on the sender and receiver side. Second, since the number of the private registered buffers is limited, the sender has to perform flow-control to make sure the receiver buffers are not overrun. SDP uses the buffer-copy approach only for transferring small messages to avoid being penalized by the message-copy overheads.

2.2 Overview of iWARP

The Internet Wide Area RDMA Protocol (iWARP) is a new initiative by the Internet Engineering Task Force (IETF) [1] and the Remote Direct Memory Access Consortium (RDMAC) [3]. The iWARP standard, when offloaded on to the network adapter, provides two primary extensions to regular Ethernet: (i) it exposes a rich interface including zero-copy, asynchronous and one-sided communication primitives and (ii) it internally relies on an implementation of the TCP/IP stack to allow such communication while maintaining backward compatibility with existing TCP/IP. iWARP comprises three protocol layers atop TCP/IP: (i) RDMAP verbs, (ii) Remote Direct Data Placement (RDDP) protocol and (iii) Marker PDU Aligned (MPA) protocol.

RDMAP verbs [6] is a thin interface that allows applications to interact with RDDP. RDDP provides reliable, in-order delivery using a reliable IP based protocol such as TCP. It distinguishes iWARP from other high-speed network stacks based on its capability to decouple data placement and message delivery; that is, even if packets arrive out-of-order, RDDP directly places them in the appropriate location of the final destination buffer (data placement), and the upper layer is informed about the placement of the data only after the entire message is placed (data delivery). This, of course, assumes that RDDP can correctly

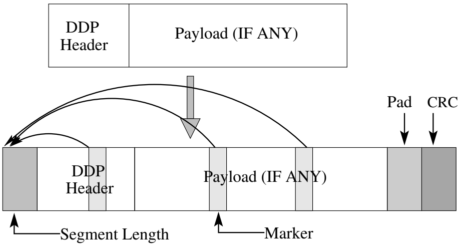


Fig. 1. MPA Protocol Frame

identify and understand the contents of out-of-order TCP/IP packets. The Marker PDU Aligned (MPA) protocol provides RDDP with the necessary support for achieving this.

Switches that support splicing [18] (e.g., firewalls and port-forwarding switches) can cause *middle box fragmentation*, i.e., packets going into the switch can be segmented into multiple packets or multiple packets can be coalesced into a single packet. This makes it impossible for the end node to recognize the RDDP headers without additional information if packets arrive out of order. To tackle this problem, iWARP uses MPA [20]. The MPA frame format, referred to as a Framing Protocol Data Unit (FPDU), is represented in Figure 1. Apart from additional headers and footers, the FPDU introduces strips of data, known as *markers*, that are spaced uniformly based on the TCP sequence number. These *markers* always point to the RDDP header and provide the receiver with a deterministic way to find them. When a packet arrives out-of-order, it can use these markers to identify the *start* of the iWARP frame and, using that, the rest of the fields.

3 Hybrid Hardware-Software iWARP Stack

Several different implementations of iWARP exist, including complete software implementations [9,21], complete hardware implementations [19] and hybrid hardware-software implementations [8]. In general, hardware implementations are optimized for performance but do not offer many advanced features; software implementations tend to be more feature complete with respect to their capability to efficiently handle out-of-order communication, packet drops, etc., but do not provide the best performance. The hybrid hardware-software implementation takes the best of both worlds by achieving high performance using network hardware accelerators, while still providing the advanced features using the capabilities of host processors. In this section, we present a high-level description of our previous work on a hybrid hardware-software iWARP stack [8].

The iWARP protocol layers perform various tasks corresponding to data ordering, data integrity, connection management, and backward compatibility. Of these, three tasks are of particular importance as they can heavily impact the performance of the stack: (i) CRC-based data integrity, (ii) connection demultiplexing, and (iii) placement of markers.

CRC is easily the most compute intensive task in the iWARP stack. There have been several attempts to improve its performance [32,16], often at the cost of additional memory usage. However, its computational overhead is still considered to be very high [26]. Thus, a complete software implementation can be heavily impacted by this overhead.

Traditional TCP/IP performs demultiplexing (DEMUX) of packets in host-space, i.e., the NIC hands over all packets to the host and the host identifies the connection to which each packet belongs and places it in the appropriate queue. While this is not a major concern for applications that only deal with a single (active) connection, this introduces significant overheads for applications dealing with several connections simultaneously (e.g., cache thrashing and CPU interruption for non-critical data). Again, doing this in software is not the ideal solution either.

Placement of markers is one of the trickiest components in the iWARP stack. Since the markers have to be inserted within the data stream, data has to be moved to achieve this. In a software implementation of iWARP, this is done by performing an additional copy of the data. This task is difficult to implement efficiently in hardware without using true scatter/gather DMA engines, which are not commonly available (most DMA engines provide a scatter/gather DMA API, but internally perform individual DMA operations). Thus, hardware iWARP achieves sub-optimal performance for this component [8].

Hybrid iWARP, behaves like software iWARP for the placement of markers (that is, it does this by performing an additional data copy), while using hardware accelerators for the remaining tasks (such as CRC and DEMUX). Thus, in summary, software iWARP performs everything in software, hardware iWARP performs everything in hardware, and hybrid iWARP performs everything in hardware except the placement of markers, which is done in software using an extra buffer copy.

4 SDP for Hybrid Hardware-Software Network Stacks

As briefly described in Section 2.1, existing designs of SDP have been heavily optimized for hardware offloaded protocol stacks. However, such designs are often not the best when utilized on hybrid network stacks. In this Section, we present a few sample existing designs that perform sub-optimally on hybrid iWARP network stacks, and propose enhancements that can improve their performance.

4.1 Redundant Buffer Copy

SDP performs data buffering for small messages. Such buffering has several advantages on hardware-offloaded network stacks including avoiding registration cost, and avoiding synchronization between the sender and receiver. However, on hybrid network stacks, these designs are redundant. For example, the hybrid iWARP stack internally performs data buffering before communication while handling the placement of markers in software. Thus, buffering at both layers is not required and causes performance overhead.

However, avoiding such redundancy is not trivial. Buffering performed within the iWARP stack allows the iWARP implementation to add markers within the data stream; data is copied such that small gaps are left open where the markers can be placed once the copy is complete. On the other hand, buffering within the SDP implementation allows it to handle the socket stream semantics where one large message sent by the sender can be read as multiple small messages by the receiver. Since iWARP follows message-based semantics, it does not allow for such capabilities. Thus, both stacks have specific purposes for buffering that cannot be ignored.

In our approach, we allow the SDP and iWARP stacks to have integrated data buffering. Specifically, the SDP stack performs buffering, but does so in a manner that is compatible with iWARP's buffering. That is, it copies data while leaving small gaps based on the TCP sequence numbers of the data (retrieved from the iWARP stack). The iWARP stack uses this buffering performed by SDP and adds the markers in-place directly in the SDP buffers. While this approach requires close interaction between the SDP and iWARP stacks, and thus loses some amount of generality of the SDP stack, it can reduce the amount of buffering required and thus improve performance.

4.2 Protocol Interface Extensions for Message Coalescing

Message coalescing has been shown to achieve high performance by reducing the number of I/O bus and network transactions required for transferring data [7]. However, it is quite difficult to achieve in hardware-offloaded protocol stacks owing to the hardware-design complexity and resource requirement associated with such a design. For hybrid network stacks, on the other hand, this might not be a concern when implemented in software using the host-memory resources. The issue, however, is that most protocols (including iWARP) do not provide any interface that allow upper layers (such as SDP) to coalesce multiple messages before sending them out on the wire. Further, message coalescing inherently suffers from issues of performance loss in cases where the sender process buffers data hoping to coalesce it with more later arriving data, while the receiver process waits for the message to be transmitted by the sender.

To solve this problem, we extended the interface provided by the hybrid iWARP implementation to allow upper layers to "append" a new message to a previously queued message whose communication has not yet been initiated. Specifically, since hybrid iWARP implementations perform flow control, communication requests that have been handed off to them might not be initiated immediately. Therefore, a later initiated communication request can append itself to this message. This approach has multiple advantages. First, multiple small messages that are being communicated in a short interval can be coalesced into one large message, thus reducing the number of network transactions and improving performance. Second, this approach does not cause any loss of performance as compared to a non-coalescing approach, since data is coalesced only when the previous message was already waiting to be sent out due to flow control; that is, a message is never artificially held back hoping to be coalesced with a later arriving message. Third, this approach reduces the number of iWARP headers that are sent out on the network since coalesced messages are sent out with one header as one single message. This is a big gain for small messages, where the iWARP header forms a major fraction of the total frame size.

4.3 Asynchronous Flow Control

Traditional implementations of SDP over hardware-offloaded iWARP perform explicit flow control. That is, if there are no credits to send data out, the sender copies the data into the temporary private buffers and *waits* for more credits to arrive (similar to advertised window in TCP). However, for hybrid iWARP implementations, such flow control is redundant since the iWARP implementation itself performs flow control as well. Furthermore, the iWARP flow control is more sophisticated as it is implemented within the kernel and uses light-weight hardware interrupts to perform asynchronous progress. Thus, in our approach, we completely disable SDP-level flow control and only rely on iWARP flow control.

While this approach works well for synchronous sockets, for asynchronous sockets, it has the drawback of its inability to call application-specific call-back functions. That is, asynchronous sockets (such as those used in Windows) allow applications to specify call-back functions that are triggered when a message send or receive is completed. To allow for such functionality, we extended the iWARP interface to specify such details, including call-back functions and message send/receive watermarks (that is, at what point the call-back should be triggered). Again, while such functionality would be extremely cumbersome and difficult to implement on hardware offloaded network stacks, it is relatively straightforward on hybrid network stacks.

5 Experimental Results and Analysis

In this section, we first evaluate our proposed approach with the latency and bandwidth micro-benchmarks in Section 5.2. We study the cache misses caused by existing approaches and how our approach reduces them in Section 5.3. Finally, we evaluate two real applications comparing our proposed approach with existing approaches in Section 5.4.

5.1 Experimental Testbed

For our experiments, we used a 4-node cluster built around SuperMicro SUPER X5DL8-GG motherboards with ServerWorks GC LE chipsets, which include 133-MHz PCI-X interfaces. Each node has two Intel Xeon 3.0 GHz processors with a 512-KB cache, a 533 MHz front-side bus and 2 GB of 266-MHz DDR SDRAM. The nodes are connected with Chelsio T110 10GE TCP offload engines through a 12-port Fujitsu XG800 switch. The software stack on the machines is based on linux-2.4.22smp and RedHat linux distribution. The driver version on the NICs is 1.2.0. For each experiment, ten or more runs/executions are conducted, the

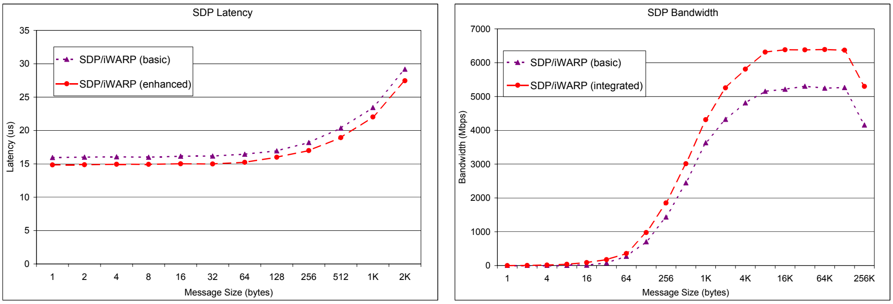


Fig. 2. SDP Performance: (a) Latency and (b) Bandwidth

highest and lowest values dropped (to discard anomalies) and the average of the remaining values is reported. For micro-benchmark evaluations, the results of each run are an average of 10,000 or more iterations.

5.2 Micro-benchmark Evaluation

Ping-pong Latency: Figure 2(a) compares the ping-pong latency of traditional SDP with our new approach. In this experiment, the sender sends a message of size S to the receiver. On receiving this message, the receiver sends back another message of the same size to the sender. This is repeated several times and the total time averaged over the number of iterations, which gives the average round-trip time. The ping-pong latency reported here is one half of the round trip time, i.e., the time taken for a message to be transferred from one node to another.

As shown in the figure, our proposed approach (SDP (enhanced)) outperforms traditional SDP (SDP (basic)) by about 10%. This is attributed to several reasons including the reduced buffer copies, and lack of redundant flow-control.

Unidirectional Bandwidth: Figure 2(b) shows a comparison of the unidirectional bandwidth. In this experiment, the sender sends a single message of size S a number of times to the receiver. On receiving all the messages, the receiver sends back one small message to the sender informing that it has received the messages. The sender calculates the total time, subtracts the one-way latency of the message sent by the receiver, and based on the remaining time, calculates the amount of data it had transmitted per unit time.

As shown in the figure, our proposed approach outperforms traditional SDP by about 20% in this case. This behavior is expected as, for large messages, traditional SDP gets significantly hurt by the additional buffer copy and loses performance. Furthermore, as we will see in Section 5.3, its performance is further affected by secondary issues such as increased cache misses.

5.3 Cache-Miss Analysis

Figure 3 shows the analysis of cache-to-network traffic ratio, comparing traditional SDP to our proposed approach; that is, how many bytes of data have to

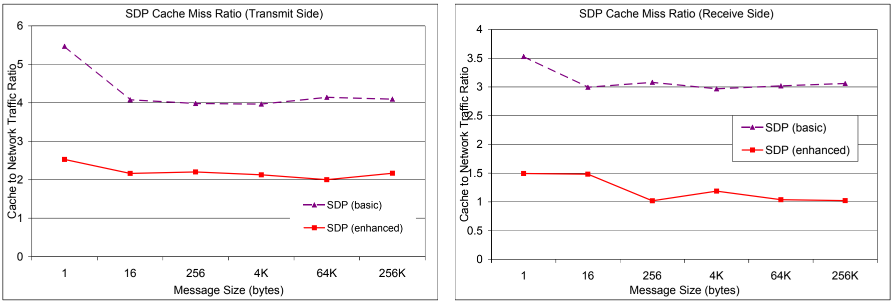


Fig. 3. SDP Cache to Network Traffic Ratio: (a) Transmit and (b) Receive

be fetched to or flushed from cache, for every byte of data sent on the network. We see that traditional SDP requires nearly four bytes of cache traffic for every byte of network traffic, as compared to our approach that requires only two.

Specifically, in the bandwidth micro-benchmark that we used, all messages are sent from the same application buffer, but the SDP and iWARP private buffers are used from a circular queue. Thus, the application buffer is always in cache, but the private buffers are never in cache. When the application data is copied to the SDP buffer, the SDP buffer needs to be fetched into cache. Next, when the data is copied from the SDP buffer to the iWARP buffer, the iWARP buffers needs to be fetched into cache. Finally, when the next set of buffers are fetched, both the SDP and iWARP buffers have to be flushed out of cache, since they are both dirty. Thus, there are two bytes fetched to cache and two bytes flushed from cache (total of four bytes of cache traffic), for every byte of data sent over the network. For our proposed approach, on the other hand, since the SDP/iWARP buffer is combined, only this combined buffer needs to be fetched into cache and flushed out from there, for a total of two bytes of cache traffic per network byte.

On the receive side (Figure 3(b)), the analysis is similar. For traditional SDP, when the data arrives, it is directly DMA'ed into the iWARP private buffer. When the data is copied to the SDP private buffer, both the iWARP and SDP private buffers need to be fetched to cache. Since the same application buffer is used throughout the experiment, it can be expected to stay in cache. However, since the SDP buffer is dirty it has to be flushed out of cache when the next set of buffers are fetched in. Thus, there are two bytes of data fetched and one byte of data flushed for every byte of data sent over the network. For our proposed approach, the combined SDP/iWARP buffer has to be fetched to cache to copy into the application buffer, i.e., one byte of cache traffic per network byte. Note that this buffer does not need to be flushed since it was never dirtied after fetching to cache.

5.4 Application-Level Evaluation

In this section, we evaluate our proposed approach based on two different applications, virtual microscope [17] and iso-surface visual rendering [13], that have been developed using the DataCutter library [11].

Overview of the DataCutter Library: DataCutter is a component-based framework [12] developed at the University of Maryland. It provides a framework, called filter-stream programming, for developing data-intensive applications. In this framework, the application-processing structure is implemented as a set of components, called *filters*. Data exchange between filters is performed through a *stream* abstraction that denotes a unidirectional data flow from one filter to another. The overall processing structure of an application is realized by a *filter group*, which is a set of filters connected through logical streams. An application query is handled as a *unit of work* (UOW) by the filter group. The size of the UOW also represents the granularity in which data segments are distributed in the system and the granularity in which data processing is pipelined. Several data-intensive applications have been designed and developed by using the DataCutter run-time framework, such as the virtual-microscope application and the iso-surface visual-rendering application.

Virtual Microscope: Virtual microscope [17] is a digitized microscopy application. The software support required to store, retrieve, and process digitized slides to provide interactive response times for the standard behavior of a physical microscope is a challenging issue [4,17]. The main difficulty stems from the handling of large volumes of image data, which can range from a few hundreds of megabytes to several gigabytes. At a basic level, the software system should emulate the use of a physical microscope, including continuously moving the stage and changing magnification. The processing of client queries requires projecting high-resolution data onto a grid of suitable resolution and appropriately composing pixels mapping onto a single grid point.

Iso-surface Visual Rendering: Iso-surface rendering [22] is a widely used technique in many areas, including environmental simulations, biomedical images, and oil reservoir simulators, for extracting and simplifying visualization of large datasets within a 3D volume. In this paper, we utilize a component-based implementation of such rendering [13].

Evaluating the Applications: Figure 4 shows the performance of the virtual microscope and iso-surface visual-rendering applications for the different SDP designs. The applications were executed with a UOW of 1KB and 8KB, respectively. The complete dataset is about 1 GB in size and is hosted on a *RAM disk* in order to avoid disk fetch overheads in the experiment. The virtual-microscope application used five filters: *read data*, *decompress*, *clip*, *zoom*, and *view*. The iso-surface visual-rendering application used four filters: *read dataset*, *iso-surface extraction*, *shade and rasterize*, and *merge/view*. Each filter performs some computation and communicates the processed data to the next filter. Once the communication is initiated, the filter starts computation on the next UOW, thus attempting to overlap communication with computation.

For the virtual-microscope application, as shown in Figure 4(a), our proposed approach outperforms traditional SDP by nearly 5%. This benefit is mainly attributed to the benefits of message coalescing. Since the UOW size used in this application is quite small, the buffer-copy overhead would not be too high.

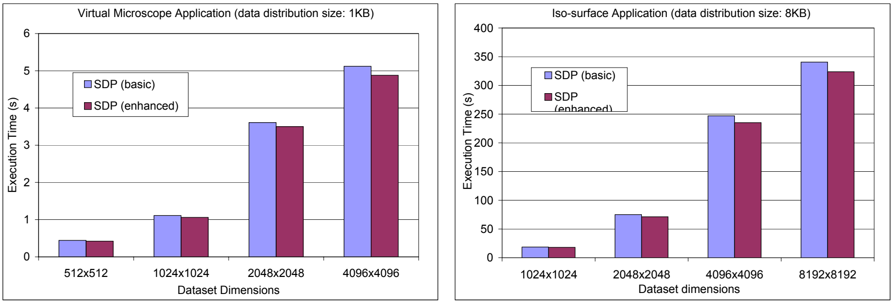


Fig. 4. Application Performance: (a) Virtual Microscope and (b) Iso-surface Oil Reservoir Data Visualization

Similarly, since after coalescing, the number of messages is fewer, running out of buffer credits happens rarely, and hence flow-control does not play a major role either.

As shown in the Figure 4(b), for the iso-surface application, our proposed approach outperforms traditional SDP by more than 5%. This benefit is attributed to mainly the reduction in buffer copies and the lack of redundant flow-control. Message coalescing would likely have little effect since the virtual microscope application uses about 8KB data chunks (UOW is 8KB), where the bandwidth is already close to the peak and coalescing would not help it much. Also, DataCutter relies only on synchronous sockets, so asynchronous sockets optimizations would not help either.

6 Conclusions and Future Work

In this paper, we proposed an extended design for SDP that uses information on which components of the network protocol stack are implemented in hardware and which are implemented in software to optimize its execution. We compared our proposed approach with existing implementations and showed that we can achieve significant performance improvements. As a part of our future work, we would like to study such enhancements in other protocol stacks, including MPI, as well. Furthermore, we would like to generalize our model so that all upper-layer protocols can query for which components are implemented in hardware and software in a uniform manner.

References

1. IETF, `http://www.ietf.org`
2. Mellanox Technologies, `http://www.mellanox.com`
3. RDMA Consortium, `http://www.rdmaconsortium.org`
4. Afework, A., Beynon, M.D., Bustamante, F., Demarzo, A., Ferreira, R., Miller, R., Silberman, M., Saltz, J., Sussman, A., Tsang, H.: Digital Dynamic Telepathology - The Virtual Microscope. In: Proceedings of the 1998 AMIA Annual Fall Symposium, American Medical Informatics Association (November 1998)

5. AMD Quad-core Opteron processor, http://multicore.amd.com/us-en/quadcore/
6. Bailey, S., Talpey, T.: Remote Direct Data Placement (RDDP) (April 2005)
7. Balaji, P., Bhagvat, S., Panda, D.K., Thakur, R., Gropp, W.: Advanced Flow-control Mechanisms for the Sockets Direct Protocol over InfiniBand. In: ICPP (2007)
8. Balaji, P., Feng, W., Bhagvat, S., Panda, D.K., Thakur, R., Gropp, W.: Analyzing the Impact of Supporting Out-of-Order Communication on In-order Performance with iWARP. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829. Springer, Heidelberg (2007)
9. Balaji, P., Jin, H.W., Vaidyanathan, K., Panda, D.K.: Supporting iWARP Compatibility and Features for Regular Network Adapters. In: RAIT (2005)
10. Balaji, P., Narravula, S., Vaidyanathan, K., Krishnamoorthy, S., Wu, J., Panda, D.K.: Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial? In: ISPASS 2004 (2004)
11. Beynon, M., Kurc, T., Sussman, A., Saltz, J.: Design of a framework for data-intensive wide-area applications. In: HCW (2000)
12. Beynon, M.D., Kurc, T., Catalyurek, U., Chang, C., Sussman, A., Saltz, J.: Distributed Processing of Very Large Datasets with DataCutter. In: Parallel Computing (October 2001)
13. Beynon, M.D., Kurc, T., Catalyurek, U., Saltz, J.: A Component-based Implementation of Iso-surface Rendering for Visualizing Large Datasets. Report CS-TR-4249 and UMIACS-TR-2001-34, University of Maryland, Department of Computer Science and UMIACS (2001)
14. Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., Su, W.K.: Myrinet: A Gigabit-per-Second Local Area Network. In: IEEE Micro 1995 (1995)
15. Boppana, R.V., Chalasani, S.: Fault-Tolerant Wormhole Routing Algorithms for Mesh Networks. IEEE Transactions on Computers, 848–864 (July 1995)
16. Herrmann, S., Castagnoli, M., Brauer, G.: Optimization of cyclic redundancy-check codes with 24 and 32 paritybits. IEEE Transactions on Communication (1993)
17. Catalyurek, U., Beynon, M.D., Chang, C., Kurc, T., Sussman, A., Saltz, J.: The Virtual Microscope. IEEE Transactions on Information Technology in Biomedicine (to appear, 2002)
18. Cohen, A., Rangarajan, S., Slye, H.: On the Performance of TCP Splicing for URL-aware Redirection. In: USENIX 1999 (1999)
19. Chelsio Communications, http://www.chelsio.com
20. Culley, P., Elzur, U., Recio, R., Bailey, S.: Marker PDU Aligned Framing for TCP Specification (November 2002)
21. Dalessandro, D., Devulapalli, A., Wyckoff, P.: Design and Implementation of the iWARP Protocol in Software. In: PDCS 2005 (2005)
22. Gao, J., Shen, H.: Parallel view dependent isosurface extraction using multi-pass occlusion culling. In: ACM SIGGRAPH (2001)
23. NetEffect Inc., http://www.neteffect.com/product-features.html
24. InfiniBand Trade Association, http://www.infinibandta.org/
25. Intel Core 2 Extreme quad-core processor, http://www.intel.com/products/processor/core2xe/qc_prod_brief.pdf
26. Khosravi, H.M., Foong, A.: Performance Analysis of iSCSI and Effect of CRC Computation. In: BEACON 2004 (2004)
27. Myricom. Myrinet home page, http://www.myri.com/

28. Petrini, F., Feng, W.C., Hoisie, A., Coll, S., Frachtenberg, E.: The Quadrics Network (QsNet): High-Performance Clustering Technology. In: Hot Interconnects (2001)
29. Pfister, G.F., Norton, V.A.: Hot-spot Contention and Combining in Multistage Interconnection Networks. IEEE Transactions on Computers 34, 943–948 (1985)
30. Qlogic Corporation, `http://www.qlogic.com`
31. Recio, R., Culley, P., Garcia, D., Hilland, J., Metzler, B.: An RDMA protocol specification (April 2005), `http://www.ietf.org/internet-drafts/draft-ietf-rddp-rdmap-04.txt`
32. Sarvate, D.V.: Computation of cyclic redundancy checks via table look-up. Communications of the ACM 31 (1998)
33. Vishnu, A., Koop, M., Moody, A., Mamidala, A., Narravula, S., Panda, D.K.: Hot-Spot Avoidance With Multi-Pathing Over InfiniBand: An MPI Perspective. In: CCGrid (2007)

Making a Case for Proactive Flow Control in Optical Circuit-Switched Networks

Mithilesh Kumar[1], Vineeta Chaube[1], Pavan Balaji[2], Wu-Chun Feng[1], and Hyun-Wook Jin[3]

[1] Dept. of Computer Science, Virginia Tech
{mithil,vineetac,feng}@cs.vt.edu
[2] Mathematics and Computer Science Division
Argonne National Laboratory
balaji@mcs.anl.gov
[3] Dept. of Computer Sc. and Engg., Konkuk University
jinh@konkuk.ac.kr

Abstract. Optical circuit-switched networks such as National LambdaRail (NLR) offer dedicated bandwidth to support large-scale bulk data transfer. Though a dedicated circuit-switched network eliminates congestion from the network itself, it effectively “pushes” the congestion to the end hosts, resulting in lower-than-expected throughput. Previous approaches either use an ad-hoc proactive approach that does not generalize well or a sluggish reactive approach where the sending rate is only adapted based on synchronous feedback from the receiver.

We address the shortcomings of such approaches using a two-step process. First, we improve the adaptivity of the reactive approach by proposing an *asynchronous*, fine-grained, rate-based approach. While this approach enhances performance, its limitation is that it is still reactive. Consequently, we then analyze the predictive patterns of load on the receiver and provide strong evidence that a proactive approach is not only possible, but also necessary, to achieve the best performance in dynamically varying end-host conditions.

Keywords: rate-based protocol, circuit switched, optical networks, LambdaGrid.

1 Introduction

Rapid advances in optical networking are producing increases in bandwidth that have outpaced the geometric increase in semiconductor chip capacity, as predicted by the Moore’s law. This means that the burden is now on the end points of communication networks to process the high bandwidth data being delivered by the network. This trend is more prominent in circuit-switched optical networks, like those found in LambdaGrids [4,12].

A LambdaGrid is a distributed grid of resources that consists of dedicated high-bandwidth optical networks, computing clusters, and data repositories. Such a distributed supercomputer will enable scientists and engineers to analyze,

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 491–502, 2008.

correlate, and visualize extremely large and remote datasets on-demand and in real time. The dedicated high-bandwidth optical networks found in LambdaGrids translate into no internal network congestion and result in pushing congestion to the end hosts. Thus, the efficiency of a network protocol at an end host is greatly influenced by its ability to adapt its transmission rate to dynamic network and endpoint conditions, thereby minimizing packet loss and maximizing network throughput.

Currently, the computational power of an end host is not sufficient to handle the high network bandwidth that is available in a dedicated circuit-switched network. This puts a cap on the maximum throughput that can be achieved. Additionally, the computational power per process is effectively reduced if there are several processes running on the end host that are competing for CPU resources. Hence, we need to understand end-host contention and come up with effective rate-adaptation techniques, which are critical when networks are fast enough to push congestion to the end hosts.

In this paper, we address the shortcomings of existing approaches in two steps. First, we present an asynchronous, fine-grained, rate-control approach that solves some performance issues but is still *reactive* in nature. Next, we present our observations from a series of experiments and analyze the predictive patterns of load on the receiver node. Based on our analysis, we provide evidence that a proactive approach is possible and required in such environments to achieve the best performance in dynamically varying end-host conditions.

2 Background

A rate-based approach is one where a constant sending rate is negotiated between a sender and receiver. Rate-based protocols perform well for high bandwidth-delay product networks. Reliable Blast UDP (RBUDP) [7] is an example of such a protocol in which the sender transmits data using UDP packets at a rate specified by the user. At the end of data transmission, the receiver sends an acknowledgment via a bitmap of missing packets. The sender then re-transmits the missing packets. The process continues until all packets are received. The mechanism is aggressive and provides reliability, but it is not adaptive to packet loss.

RAPID [1] is an end-host aware, rate-adaptive protocol, where rate adaptation is based on proactive feedback from the receiver. The receiver is monitored by a soft real-time process that attempts to guess the time and duration when the receive process is context-switched and replaced by another process. It then informs the sender, which suspends transmission to avoid packet loss when the receiver process is context-switched and suspended. Although being a proactive approach, a major drawback of this protocol is that it is difficult to predict the exact time when the receive process will be suspended. It is even more difficult to match the sender's transmission suspension with the receive process' suspend interval, especially because the two nodes are physically separated by a reasonable amount of network delay [2]. Moreover, stopping data transmission completely is a drastic step to take when we aim at keeping the network utilization high.

Another RBUDP variant, RBUDP+ [3], uses the same scheme but a different estimate of the time and duration for which the receive process has been rescheduled. Like RAPID, the prediction of time and duration is almost impossible.

An enhancement of RAPID and RBUDP+ is RAPID+ [2], which focuses on dynamically monitoring the packet loss at the receiving end host, so that it can be used to adapt the sending rate. Accurate prediction of when packet loss would occur increases performance and circuit utilization. However, in this case, rate adaptation is initiated only after most of the losses have already occurred.

The Simple Available Bandwidth Utilization Library (SABUL) [6] is a rate-based as well as window-based protocol that has been designed for data-intensive applications over a shared network. Its delay and window-based congestion control makes it TCP-friendly but brings along the same characteristics of TCP that make it inefficient when congestion has been pushed to the endpoints.

The Group Transport Protocol (GTP) [15] is a receiver-driven protocol that performs well in multipoint-to-point and multipoint-to-multipoint environments and ensures fairness between different connections on the same end host. Like other rate-based protocols, SABUL and GTP wait for packet loss to occur before providing feedback to the sender to transmit with revised sending rates.

TCP does not perform well for large bandwidth-delay product networks, as found in LambdaGrids, because of the overhead involved in congestion and flow control. In order to improve TCP performance, significant research has been done to improve TCP congestion control [9,10,13,16]. Other complementary research has been done to improve flow control [5,11,14]. However, none of these improvements or variants of TCP were designed for networks with nearly zero congestion. In this paper, we use UDP because it is lighter and faster and can be enhanced to provide reliability without worrying about network congestion.

3 Asynchronous Fine-Grained Rate Control

In this section, we introduce our fine-grained, rate-based control protocol called ASYNCH and compare its performance with existing rate-based protocols.

3.1 Basic Idea

Many existing rate-based protocols such as RAPID+, SABUL and GTP use *reactive* rate control to avoid end-host congestion, where the sending rate is varied *after* an event, such as a packet drop, occurs. Further, these approaches use a coarse-grained feedback mechanism. Specifically, they utilize multi-round communication; in the first round, the sender attempts to send data at the maximum rate. If there are dropped packets in the second round, these dropped packets are sent at a slower pace. These rounds continue till all the data has been successfully communicated.

While such an approach is simple, it has several performance implications. First, a coarse-grained feedback approach, such as that described above, has a high overhead of inaccuracy. For example, if a receiver is only capable of receiving data at 5 Gbps, a sender transmitting a 10-GB file at 10 Gbps would end up dropping half the packets (5 GB). The sender, however, would receive feedback about its high sending rate only at the end of the first communication round, i.e., after the entire file has been sent out. In the second round, the remaining 5 GB has to be retransmitted. Thus, the inaccurate sending rate (in this case, 10 Gbps) can result in a major loss of performance, which is expected to further worsen as the amount of data being communicated increases.

Second, a reactive approach is fundamentally limited in its ability to handle dynamic environments where the receiving end-host is executing other processes. By the time the sender receives feedback for rate adaptation, the receiver's capability to receive data might have already changed.

In order to analyze the behavior of the receiver end-host and its dynamics, we implemented an asynchronous, fine-grained, reactive rate-control protocol named ASYNCH. This protocol, while still reactive, addresses the issue of coarse-grained feedback; it asynchronously sends feedback to the sender at regular intervals of time, instead of waiting for the entire file to be transferred before sending the feedback. This mechanism allows feedback to be independent of the file size. On the other hand in synchronous feedback, such as in RAPID+, the receiver sends feedback only at the end of each communication round.

In ASYNCH, upon receiving feedback, the sender calculates the current loss rate. If an increase in loss rate (compared to the last calculated value) is detected, the sending rate is decreased. The sender considers at least two feedback messages from the receiver for deciding what kind of rate adaptation is required. If a high loss rate has been observed in only one round, it is treated as a temporary loss, and no action is taken. If the receiver does not experience any packet loss for k successive rounds, then the sender will increase its sending rate. This reactive rate adaptation is expected to reduce any further loss at the receiver.

3.2 Performance Evaluation

We evaluate the performance of ASYNCH and RAPID+ for various conditions on the receiver end-host and analyze the results. We first show how throughput degrades when the sending rate increases beyond the receiver's capacity to receive. We then explain the role played by the *round-trip time* (RTT) latency and the receiver's socket buffer size on packet loss rate and network throughput.

Methodology. The testbed is a three-node sender-receiver setup with the middle node acting as a wide-area network (WAN) emulator using Netem [8]. A file size of 1GB, RTT of 56ms and MTU of 9000 bytes was used for all experiments, unless otherwise specified. The choice for this setup and configuration is based on our attempt to emulate a real dedicated circuit-switched, long-fat network. In order to obtain consistent results, we bind the receive process to the same core for our experiments. The system configuration is summarized in Table 1.

High Network Speed. One of the primary reasons for packet drops is the discrepancy in the protocol processing requirements for data sending and receiving. Specifically, data transmission with TCP/IP or UDP/IP is typically a

Table 1. System Configuration

Processor	Dual-core AMD Opteron 2218
Cache Size	1024 KB
RAM	4 GB
Network Adaptor	Myrinet 10Gb
Kernel	2.6.18

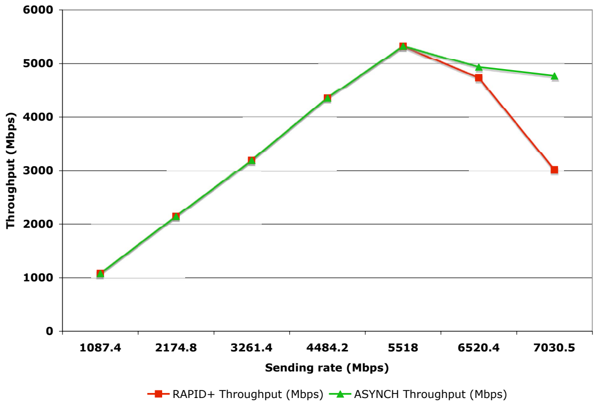


Fig. 1. Network throughput degrades after the sending rate reaches an optimal point

lighter weight operation compared to data reception, owing to various optimizations such as integrated checksum and copy and the lack of data demultiplexing requirements that are necessary on the receiver side. In our experiments with a 10 Gbps network, the sender, for example, is able to transmit packets at 7 Gbps. However, the receiver is only able to receive data at 5.5 Gbps.

Figure 1 demonstrates this behavior. For sending rates less than 5.5 Gbps, the achieved throughput is about the same as the sending rate, and there is no packet loss. Beyond 5.5 Gbps, however, there is a sharp rise is the packet loss rate, resulting in a decline in throughput. Unlike RAPID+, ASYNCH utilizes a fine-grained feedback mechanism to adapt its rate quickly resulting in up to 58% better throughput as compared to RAPID+. For a sending rate of 7 Gbps, the loss rate for ASYNCH is only 9.5% compared to the 21.49% for RAPID+.

For the rest of the experiments, we are interested in studying the capabilities of the receiver end-host; therefore we use a peak sending rate of 5.5 Gbps.

Socket Buffer Size of the Receiver. UDP/IP communication takes place through socket buffers. Data received is stored in the socket buffer until the application reads it. Thus, while a large socket buffer can provide more tolerance to a receiver's slow receiving capability, it can result in memory wastage. Accordingly, an optimal buffer size needs to be chosen to balance memory wastage and packet loss.

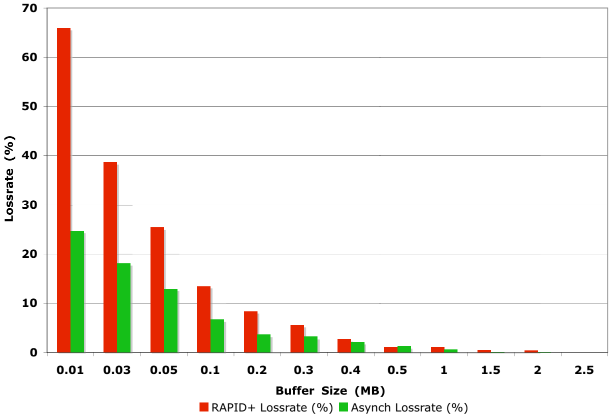


Fig. 2. Effect of socket buffer size on loss rate

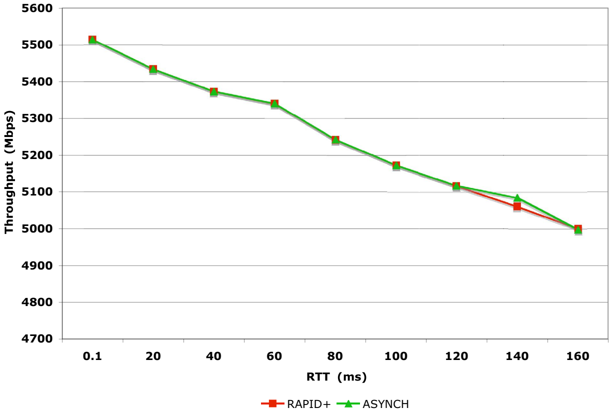


Fig. 3. Effect of RTT on throughput

Figure 2 shows the effect of the socket buffer size on loss rate. For very small buffer sizes (10-100 KB), substantial packet loss occurs, resulting in poor throughput. For buffer sizes larger than 100 KB, ASYNCH's loss rate drops to about 5-7% or less. While the loss rate for RAPID+ decreases with the socket buffer size as well, we notice that ASYNCH's loss rate is much smaller than that of RAPID+ for all buffer sizes.

Round-Trip Time. Unlike TCP, in the case of UDP data transfers, RTT does not play a significant role in UDP throughput due to the lack of congestion- and flow- control mechanisms in UDP. However, to achieve reliability, both RAPID+ and ASYNCH use a TCP control channel; the sender waits for packet-loss feedback from the receiver. This feedback mechanism, however, directly depends on the RTT and causes the throughput to drop with increasing RTT. Figure 3 illustrates this relationship. That is, for both RAPID+ and ASYNCH, the throughput drops by about 65 Mbps for every 20-ms increase in RTT.

4 A Case for Proactive Rate Control

Here we analyze the receiver end-host in environments where a number of competing processes are scheduled on the receiver node.

4.1 Effect of Load on the Receiver End-Host

In environments where a number of competing processes are scheduled, the receiver's network process must compete with the other processes on the end host for CPU resources. The competing processes may be of various types and can be I/O-bound or CPU-bound.

We use four types of loads for our experiments. The first is a purely computational workload that runs entirely in user space. The second is a system-call load that reads and writes data to a flat file. The third is a memory-intensive load that reads data from a 1GB buffer. Finally, we have a network load that acts as a forwarder for packets received by the receiver process. We use a four-node setup; two of the nodes perform the actual communication, one node acts as a delay node emulating a high-latency network, and the fourth node acts as a gateway, where the receive and forward processes are running. The gateway node performs some computation on every received packet and then forwards it to the final recipient of the data.

With the exception of network load, all loads are running on the same processor core as the receive process. For network load, we run the receiver on P1C2 (Processor 1, Core 2) and all forwarder threads on P1C1 since cores on the same processor share cache and memory. This helps obtain maximum end-to-end throughput.

Figures 4 and 5 compare the loss rate and throughput of RAPID+ and ASYNCH with increasing load, respectively. For the purely computational load, ASYNCH quickly adapts its sending rate, thus avoiding any severe packet loss. However, neither the loss rate nor throughput has any fixed pattern with increasing load.

For the system-call load, the average loss rate is much higher for both ASYNCH and RAPID+. This is attributed to the higher priority assigned by the operating system to system-call workloads. Consequently, fewer CPU resources are allotted to the receiver process, resulting in an increased loss rate. On the other hand, the memory-intensive load demonstrates behavior that is better than the system-call workload, but worse than the compute workload.

For the network load, the throughput, as seen by receiver process, is not affected significantly by the increase in number of forwarder threads, mainly because the forwarder is running on a separate core. However, the end-to-end throughput (not shown in figure) reduces, since the forwarder threads, all running on the same core, compete amongst themselves for CPU resources.

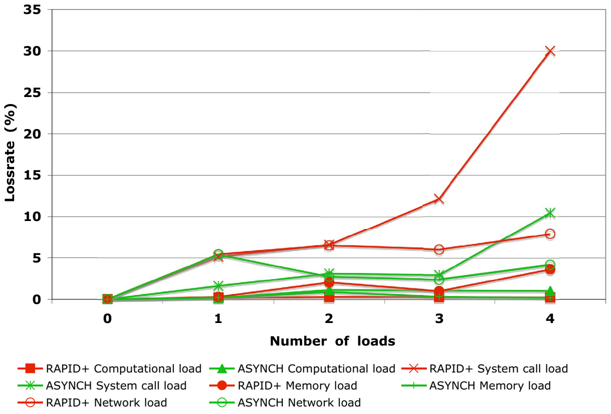


Fig. 4. Effect of various loads on loss rate

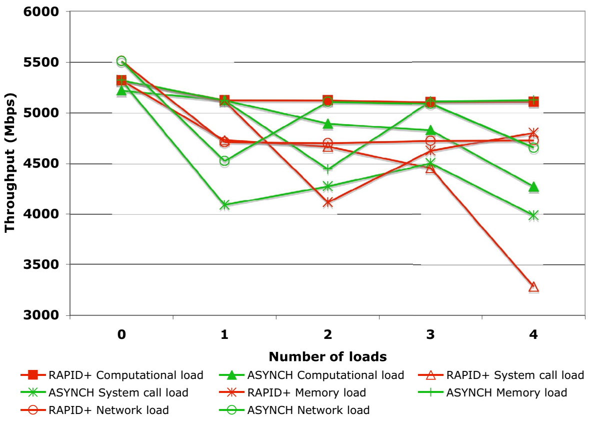


Fig. 5. Effect of various loads on throughput

4.2 Loss Patterns

Figure 6 shows the pattern of packet loss when data is transmitted with rate adaptation disabled. We disable rate adaptation in this experiment in order to observe the loss patterns that would help us understand if the current reactive approaches are able to adapt precisely during intervals of packet losses.

The sharp spikes for pure computational load reveal that all packets in a small interval of time are lost. Similar spikes were observed for system-call load and memory-intensive loads, although not shown in the figure. When rate adaptation is enabled, ASYNCH and RAPID+ interpret the spike as a heavy loss rate and wrongly adapt the sending rate. In general, the presence of spikes in any loss pattern is likely to convey a wrong signal to the reactive rate-adaptation algorithm. The loss pattern for a network load does not have any spikes and

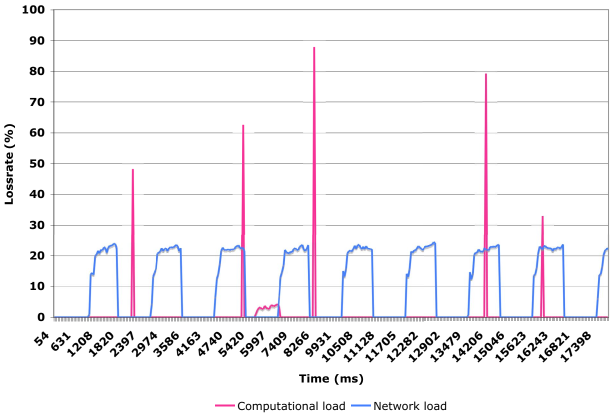


Fig. 6. Loss patterns for computation load and network load

flattens on the top. Thus, the reactive rate adaptation is expected to work for this case as the future loss pattern is likely to remain the same as the current loss pattern.

4.3 Reactive Versus Proactive Approach

Based on the loss patterns described above, it is clear that a reactive approach is not suitable to adapt the rate, based on dynamic conditions at the receiver end host. A reactive approach will work if the conditions at the receiver are static, resulting in steady loss patterns, e.g., loss patterns due to another network load on the receiver. On the other hand, a proactive approach can potentially predict

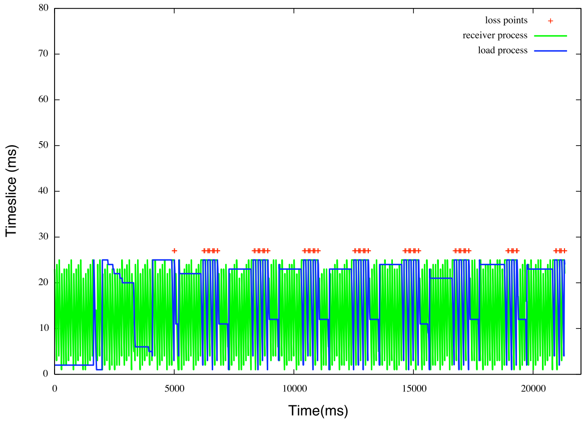


Fig. 7. Timeslice consumption of processes showing intervals when each of them is currently scheduled for execution

the time when a loss event will occur and take necessary action to prevent packet drops. However, designing a proactive approach is a difficult problem that requires understanding of the way an operating system scheduler handles processes.

Figure 7 shows the timeslices awarded to the receive process and a memory-intensive load process. The exact times when packet loss occurs have been marked in the figure. These points are always located in the same time interval when the load process is running, depriving the network process of CPU resource and thus causing it to drop packets. In other words, packet losses occur exclusively because the receive process gets rescheduled and is replaced by the competing load process.

4.4 A Proactive Approach

In designing a proactive approach, we need to estimate in advance when the receive process will be rescheduled and replaced by another process for execution. We will refer to this time as *context-switch time*, since this rescheduling essentially involves a context-switch. There are two approaches for predicting when a context-switch for the receive process will occur, as discussed below.

Polling Dynamic Priority. Polling the dynamic priority of the receive process to estimate the context-switch time has been used by Banerjee et al. [1]. The idea is to note the dynamic priority of the receive process when a loss actually occurred. During polling, if the dynamic priority reaches one less than the previously noted value, the sender is notified to suspend transmission for an amount proportional to the average sleep time of the process. However, as seen in Figure 8 for a memory-intensive load, the dynamic priority of the receive process takes only three values and just prior to getting context switched, there are no changes to the dynamic priority value. This behavior has been observed for other kinds of loads as well and therefore this approach is not reliable.

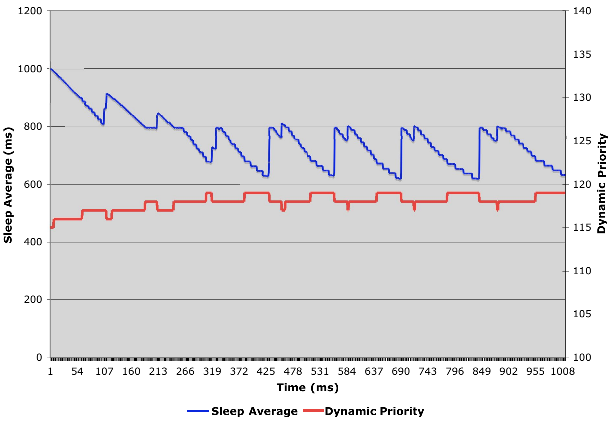


Fig. 8. Sleep average and dynamic priority for a memory access intensive process

Polling Sleep Average. Figure 8 shows the average sleep time of the receive process. We see that the average sleep time of a process follows a saw-tooth pattern. A context-switch happens whenever the average sleep time has reached a local minimum. Because of this uniform and periodic pattern, it is possible to estimate the time of an upcoming context-switch.

If at any instance of time, we have the maximum and minimum *average sleep time* (given by MAX_SLEEP_AVG and MIN_SLEEP_AVG), we know that the process started its execution with its *average sleep time* $= MAX\_SLEEP\_AVG$ and will be rescheduled when its *average sleep time* has reached MIN_SLEEP_AVG. We take an action when the *average sleep time* reaches $(MAX\_SLEEP\_AVG + MIN\_SLEEP\_AVG)/2$. We must constantly update the maximum and minimum values, since they are likely to change.

5 Conclusion and Future Work

In this paper, we presented an asynchronous, feedback-based, reactive, rate-control protocol called ASYNCH that features a fine-grained rate-control mechanism. Our protocol effectively solves some of the problems faced by current rate-based protocols that adapt sending rate, leading to accurate rate adaptation and therefore higher throughput. We also analyzed the end-host behavior in dynamic environments and made a case for a proactive protocol, which is more suitable for handling such environments.

References

1. Banerjee, A., Feng, W., Mukherjee, B., Ghosal, D.: RAPID: An End-System Aware Protocol for Intelligent Data Transfer over LambdaGrids. In: 20th IEEE International Parallel and Distributed Processing Symposium, 10 pages (2006)
2. Datta, P., Feng, W., Sharma, S.: End-System Aware, Rate-Adaptive Protocol for Network Transport in LambdaGrid Environments. In: ACM/IEEE Supercomputing 2006 (2006)
3. Datta, P., Sharma, S., Feng, W.: A Feedback Mechanism for Network Scheduling in LambdaGrids. In: IEEE International Symposium on Cluster Computing and the Grid (2006)
4. DeFanti, T., Laat, C., Mambretti, J., Neggers, K., Arnaud, B.: TransLight: A Global-Scale LambdaGrid for e-Science. Communications of the ACM, 34–41 (2003)
5. Fisk, M., Feng, W.: Dynamic adjustment of tcp window sizes. Los Alamos Unclassified Report 00-3321 (2000)
6. Grossman, R.L., Mazzucco, M., Sivakumar, H., Pan, Y., Zhang, Q.: Simple Available Bandwidth Utilization Library for High-Speed Wide Area Networks. J. Supercomput (Netherlands) 34(3), 231–242 (2005)
7. He, E., Leigh, J., Yu, O., Defanti, T.A.: Reliable Blast UDP: Predictable High Performance Bulk Data Transfer. In: IEEE International Conference on Cluster Computing, pp. 317–324 (2002)
8. Hemminger, S.: Network Emulation with NetEm. In: Australia's National Linux Conference (LCA 2005) (2005)

9. Hengartner, U., Bolliger, J., Gross, T.: TCP Vegas Revisited. In: IEEE INFOCOM, vol. 3, pp. 1546–1555, March 26-30 (2000)
10. Katabi, D., Handley, M., Rohrs, C.: Congestion Control for High Bandwidth-Delay Product Networks. Comput. Commun. Rev. (USA) 32(4), 89–102 (2002)
11. Semke, J., Mahdavi, J., Mathis, M.: Automatic TCP Buffer Tuning. In: ACM SIGCOMM 1998, New York, NY, USA, pp. 315–323 (1998)
12. Taesombut, N., Chien, A.A.: Distributed Virtual Computers (DVC): Simplifying the Development of High Performance Grid Applications. In: IEEE International Symposium on Cluster Computing and the Grid, 2004, pp. 715–722, April 19-22 (2004)
13. Tan, K., Song, J., Zhang, Q., Sridharan, M.: A Compound TCP Approach for High-Speed and Long Distance Networks. In: 25th IEEE International Conference on Computer Communications, vol. 25, pp. 1–12 (2006)
14. Weigle, E., Feng, W.: Dynamic Right-Sizing: A Simulation Study. Computer Communications and Networks 10, 152–158 (2001)
15. Wu, R.X., Chien, A.A.: GTP: Group Transport Protocol for LambdaGrids. In: IEEE International Symposium on Cluster Computing and the Grid, pp. 228–238 (2004)
16. Xu, L., Harfoush, K., Rhee, I.: Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks, Hong Kong, China, vol. 4, pp. 2514–2524 (2004)

FBICM: Efficient Congestion Management for High-Performance Networks Using Distributed Deterministic Routing

Jesús Escudero-Sahuquillo[1], Pedro García[1], Francisco Quiles[1], Jose Flich[2], and Jose Duato[2]

[1] Dept. of Computing Systems, University of Castilla-La Mancha, Spain
{jescudero,pgarcia,paco}@dsi.uclm.es
[2] Dept. of Computer Engineering, Technical University of Valencia, Spain
{jflich,jduato}@gap.upv.es

Abstract. As the number of components in cluster-based systems increases, cost and power consumption also increase. One way to reduce both problems is using smaller networks with adequate congestion management mechanisms. Recent successful proposals (RECN) eliminate the negative effects of congestion, the Head-of-Line (HOL) blocking, leaving congestion harmless. RECN relies on source-based networks architectures, where the entire route is placed at packet headers before injection. Unfortunately, distributed table-based routing is also common in cluster-based networks, being InfiniBand the most prominent example. We propose a novel congestion management technique for distributed table-based routing. The mechanism relies on additional congestion information located at routing tables. With this information HOL blocking is minimized by smartly using switch queues. Detailed memory organization and the way congestion information is updated/propagated is described. Preliminary results indicate that with modest resource requirements maximum network performance is kept regardless of congestion.

Keywords: High-Performance Interconnects, Congestion Control, Distributed Routing.

1 Introduction

During the last decades, the evolution of interconnection network technology has been really significant. Of course, this evolution has taken place in parallel with the proliferation of computing and communication systems based on these networks: Massive Parallel Processors, Local and System Area Networks, Clusters of PCs and Workstations, IP routers, and Networks-on-Chip. In order to work at maximum capacity, such systems demand low packets latencies and high network bandwidth. On the other hand, the popularity and utilization of these systems is constantly growing. As a clear example, more than half of the most powerful systems (top500 list)[1] are clusters. Consequently, there exists a great interest on achieving the best possible performance of the interconnection network and, in fact, researchers and designers have proposed many architectures and techniques with the aim of improving any key aspect of network functionality.

One of such aspects is the network behavior during congestion situations. These situations appear when several packet flows persistently request the same output inside a

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 503–517, 2008.

switch. In such situations, and assuming a lossless network (so, discarding packets is not allowed), packets from the involved flows arrive at the corresponding input buffers faster than they can cross to the requested output, thus these buffers finally collapse. Moreover, flow control propagates congestion throughout the network, following the reversal path of the flows contributing to congestion (thereby forming "congestion trees" across the network [2]). When congestion reaches many network points, the immediate consequence is a severe network performance degradation (throughput drops, latency increases exponentially).

The specific cause of this degradation is that congested flows may share some network resources (queues, links) with non-congested flows, thereby the former slowing the advance of the latter. In detail, in a queue storing packets belonging to congested and non-congested flows, a "congested packet" reaching the head of the queue will usually have to wait for a long period before being forwarded, and consequently all the other packets in the queue (both congested and non-congested) will suffer this delay. In general, this effect is known as Head-Of-Line (HOL) blocking, and it may limit the throughput of the switch up to about 58% of its peak value [3].

In modern high-speed interconnection networks, the use of some technique for solving the problems related to congestion has become actually mandatory, since current networks are usually designed using a low number of network components (due to the high cost and power consumption of such components), thereby decreasing the offered bandwidth and increasing congestion probability. In that sense, many proposed techniques can help to reduce the negative effects of congestion (see section 2), but none as satisfactorily as the technique known as Regional Explicit Congestion Notification (RECN) [4,5,6]. Contrary to common techniques that try to avoid or prevent congestion, RECN completely eliminates the HOL blocking produced by congested flows leaving congestion harmless. RECN requires a reduced set of additional resources per switch port, which are dynamically allocated for storing congested packets separately from non-congested ones. In this way, RECN keeps network performance at maximum during congestion situations at a moderate cost.

However, an essential requirement for any RECN implementation is the use of source-based routing where the entire path of a packet is encoded in its header. RECN identifies congested packets by comparing the explicit route stored in the header of each packet to explicit routes leading to congested points. Of course, this limitation prevents RECN from being applied in any network technology that uses table-based, distributed routing, like Infiniband [7]. Therefore, efficient congestion management on these technologies is still an open issue, while the use of such technologies is far from being unpopular.

In order to fill this significant technical gap, we present in this paper a new technique able to eliminate the HOL blocking produced by congested flows in networks implementing table-based, distributed deterministic routing. We have called this new technique Flow-Based Implicit Congestion Management (FBICM). FBICM eliminates HOL blocking following the same basic approach as RECN, so by detecting congested packets and separating them from non-congested ones. However, FBICM differs from RECN in many aspects, especially the following ones:

1. Congested points are not addressed by means of explicit routes. Instead, FBICM identifies congested points implicitly, by keeping track of the different flows passing through these points.
2. Congested packets are not identified as packets following an explicit route to a congested point, but as packets belonging to flows involved in a congested situation.
3. Congestion information is associated to destinations whereas in RECN is associated to internal network points. Thus, totally new implementations are required.

As a consequence of these three basic differences, FBICM requires new formats for congestion notifications, and new policies for congestion detection, congestion information propagation and resource management. As we will show in this paper, we have carefully considered all these aspects in the design of FBICM, in such a way that FBICM finally offers the same performance as RECN, but in a different technological context. Summing up, FBICM offers the benefits of an efficient, cost-effective congestion management to interconnection networks that implement table-based, distributed deterministic routing.

The rest of the paper is organized as follows. Section 2 shows an overview of the existing related work. Next, in Section 3, the basics of the new FBICM mechanism are presented. In Section 4, FBICM is compared in terms of performance and resource needs to previously proposed techniques which also reduce or eliminate HOL blocking. Finally, in Section 5, some conclusions are drawn.

2 Related Work

The risk of congestion in interconnection networks is a well-known problem, and many strategies have been proposed. The simplest ones are, however, overdimensioning the network and/or dropping packets in congestion situations. However, none of them are suitable for modern interconnection networks due, respectively, to the high cost and consumption of current network components and to the lossless character of these networks.

Other more elaborated techniques have been specifically proposed for avoiding or eliminating congestion. For instance, proactive strategies are based on reserving network resources for each data transmission, requiring a traffic planification based on network status [8]. However, this status information is not always available, and the resource reservation procedure introduces significant overhead. On the other hand, reactive congestion management is based on notifying congestion to the sources contributing to its formation, in order to cease or reduce the traffic injection from those sources [9]. Unfortunately, these solutions may not be efficient due to the delay between congestion detection and notification.

Other strategies deal with the HOL blocking problem, thus, indirectly dealing with congestion. Several HOL blocking elimination strategies have been proposed: Virtual Output Queues (VOQs) [10], Dynamically Allocated Multiqueues (DAMQs) [11], congestion buffers [12], etc. Most of these techniques rely on allocating different buffers for storing separately packets belonging to different flows.

In general, traditional HOL blocking elimination techniques are feasible or effective, but not feasible and effective at the same time. For instance, the use of VOQs at network level requires as many queues at each port as end-points in the network, being so an effective but not scalable technique. A variation of VOQ uses as many queues at each port as output ports in a switch [13](eliminating switch-wide HOL blocking). So, this technique is feasible, but it does not eliminate completely network-wide HOL blocking.

In contrast to these techniques, RECN eliminates HOL blocking in an efficient and scalable manner. Like VOQs, RECN tries to separate congested and non-congested flows by storing them in different queues. Specifically, RECN adds a set of additional queues (set aside queues, SAQs) to the standard queues at every input and output port of a switch. While standard queues will store non-congested packets (RECN assumes that packets from non-congested flows can be mixed in the same buffer without producing significant HOL blocking), SAQs are dynamically allocated for storing packets passing through a specific congested point. SAQs are allocated only when congestion arises, and can be deallocated when congestion vanishes, so RECN uses these resources efficiently.

Every set of SAQs is controlled by means of a CAM (content addressable memory), in such a way that each CAM line contains information for managing an associated SAQ, including the information required for addressing a congested point. In that sense, RECN assumes the use of source deterministic routing, thereby addressing congested network points by means of explicit routes toward these points. These routes can be indicated by sets of "hops" stored in the CAM lines. Once a packet arrives at a port, the routes stored in the port CAM lines can be compared to the explicit route stored in the packet header, in order to know if the packet will cross any of the detected congested points. In this way, congested packets can be easily detected and stored in the corresponding SAQ, thereby preventing them from sharing standard queues with non-congested packets, and consequently avoiding HOL blocking.

Although the RECN basic mechanism described above has proved to be very efficient, it presents the obvious limitation of requiring the use of source deterministic routing. As we have already mentioned, the main aim of the new HOL blocking elimination technique presented in this paper (FBICM) is to solve this flaw by offering the same benefits as RECN, but in a table-based, distributed deterministic routing context. The following section describes how this goal can be achieved.

3 FBICM Description

In this section we describe FBICM and its operation. Firstly we focus on the assumed switch architecture: memory organization, routing and arbitration mechanisms, etc. Next, we explain in detail the main features of FBICM: congestion detection, queue allocation, etc., illustrating the descriptions by means of operation diagrams.

3.1 Switch Architecture

In Fig. 1, a scheme of the assumed switch architecture is shown. It is important to note that FBICM, just like RECN, does not limit the number of ports of a switch. We assume that both at input and output ports, Link Controllers (LC) coordinate the transfer of flow control units at each side of the physical channel. In that sense, a credit-based flow control mechanism at packet level has been assumed.

As can be seen in the scheme, there exist RAM memories only at input ports. This is because FBICM has been developed for Input Queued (IQ) switches, thus we assume

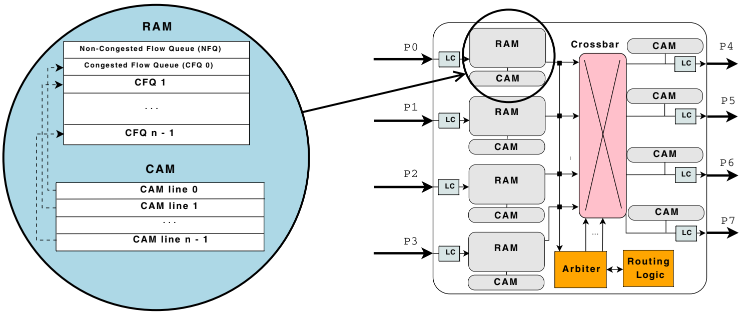


Fig. 1. Switch Architecture Assumed by FBICM (4x4 Switch)

that only switch input ports have queues. This kind of switches are very popular, and they are cheaper than Combined Input and Output Queued (CIOQ) switches, the former requiring less memory and resources for operating than the latter[1]. Input RAMs are organized in queues, dynamically managed. We assume two types of queues: non-congested flow queues (NFQs), where non-congested packets are stored, and congested flow queues (CFQs), where packets belonging to congested flows are placed. Moreover, associated with each set of CFQs, there is a content addressable memory (CAM), which contains information about the congested ports and the status of each CFQ. This port memory organization can be seen in the "zoom" include in Fig. 1.

Although output ports have neither NFQs nor CFQs, FBICM requires a CAM per output port, in order to propagate congestion information from downstream input port CAMs to upstream input port CAMs, as we will explain later.

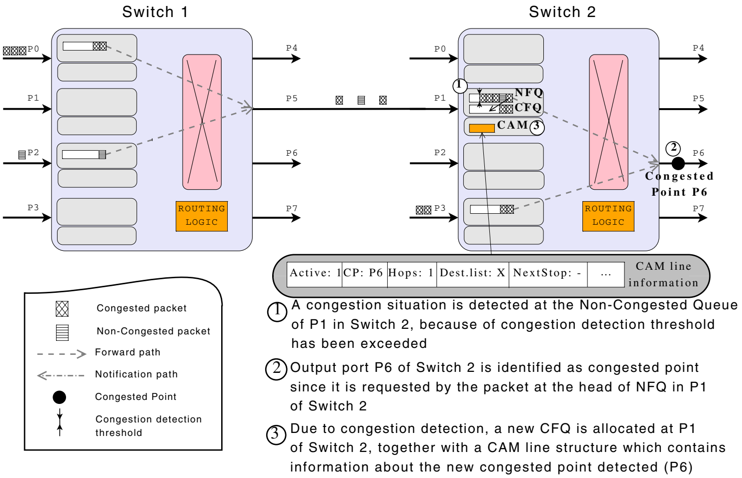
Regarding the routing mechanism, FBICM has been designed for networks using table-based, distributed deterministic routing, thus the Routing Logic is based on a routing table that indicates the output port for each incoming packet. We assume routing tables are filled during network setup according to some routing algorithm. In our experiments, we have used a self-routing algorithm for multistage networks, calculating the output port as a function of the destination of the packet. Therefore, the only routing information that packets need to include in their header is their destination (instead of a explicit, complete route).

On its side, the switch arbiter selects (following an "oldest-first" policy) routed packets for switching them through the crossbar, as long as the required input-output connection is possible (so, only if the corresponding crossbar internal path is free). The crossbar assumed for this model is a NxN multiplexed crossbar, where N is the number of input/output channels. Finally, the switching technique used for this model is Virtual Cut-Through (VCT).

3.2 Congestion Detection

A congestion situation happens when some packets request the same output port within the switch and this situation persists in time. In normal conditions the rate at which

[1] Note that the latest RECN version [6] was also designed for IQ switches.



Active: 1	CP: P6	Hops: 1	Dest.list: X	NextStop: -	...

(1) A congestion situation is detected at the Non-Congested Queue of P1 in Switch 2, because of congestion detection threshold has been exceeded

(2) Output port P6 of Switch 2 is identified as congested point since it is requested by the packet at the head of NFQ in P1 of Switch 2

(3) Due to congestion detection, a new CFQ is allocated at P1 of Switch 2, together with a CAM line structure which contains information about the new congested point detected (P6)

Fig. 2. Operation Example of Congestion Detection and Primary Queue Allocation in FBICM

packets arrive to a given input port will be approximately the same as packets leaving the output port, thus the occupation of non-congested flow queues should be low. In congestion situations, the NFQs at the switch input ports will fill, and it is necessary to detect these congestion situations as a first step for avoiding the *HOL blocking* that would appear.

FBICM detects congestion situations at input ports by enabling a detection threshold at non-congested flow queues. When the number of packets in a NFQ exceeds this threshold, a new congested point is identified. Specifically, we assume that is very likely that the packet which is allocated at the head of the NFQ belongs to a flow contributing to create congestion in the switch. Therefore, this detection method infers this packet is delaying the normal packet flow towards its requested output port and thus, this output port is probably a congested point[2], (so being the "root" of a growing congestion tree).

Actually, it is possible that the packet at the head of the NFQ is not a congested one. This would be, in fact a failure in the detection mechanism, but it will be corrected (as we will explain alter) by means of the packet post-processing and the deallocation policy of CFQs.

In Fig. 2 (pass 1), we can see a simple operation example about the situation described above. There are two types of packets: congested which are routed to P6 at switch 2, and non-congested which are routed to other destinations. The congested ones come from P0 in Switch 1 and P3 in Switch 2 and the non-congested ones come from P2 in Switch 1. Since packets from different input ports request the same output port, a contention situation appears and the normal flow of packets is delayed, causing a congestion situation. When the occupancy of the NFQ at input port P1 of switch 2 exceeds the detection threshold value (in this case the threshold value is five packets), the congestion detection takes place at this input port. Since the output port requested by the

[2] This detection method has been also used and tested in previous congestion management techniques [6], exhibiting a great accuracy.

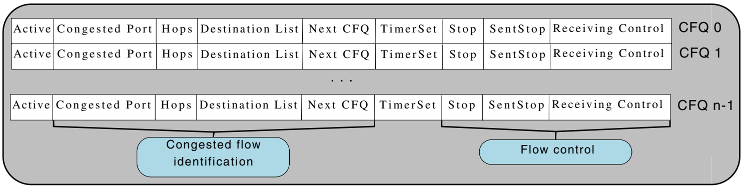


Fig. 3. CAM organization

packet at the head of the NFQ detecting congestion is P6, this output port is identified as a congested point (see Fig. 2 pass 2).

3.3 Primary CFQ Allocation

Once a switch output port is detected as a congested point, FBICM must separate incoming packets addressed to that congested point from incoming packets which do not request that port. In order to do that, a CFQ will be dynamically allocated at any input port that detects congestion, as well as an associated CAM line. The CAM line will contain the CFQ status information and also information for identifying the congested port.

Figure 3 shows in detail the organization of CAM. As can be seen, congestion information consists of four fields: Congested-Port, hops-to-reach, list-of-destinations, and NextCFQ. The Congested-Port field indicates an output port detected as congested from the input port, while the list-of-destinations field is intended to contain all the destinations that would lead incoming packets to request that congested port. In this way, given the destination of an incoming packet, it is possible to know immediately if it would request the congested port, thereby allowing to decide if it belongs to a congested flow[3] and it must be stored in the associated CFQ. Many destinations can be progressively added to the list-of-destinations, as congested packets with different destinations pass through the input port. The "hops-to-reach" field is used to measure (in hops) the distance from the CAM line input port to the congested point, thus it is only useful when congestion information is propagated upstream from the input port (see subsection 3.5). The "NextCFQ" field is used also during congestion information propagation.

On the other hand, the CFQ status information stored in each CAM line consists of four bits, indicating respectively if the CAM is active (Active bit), if the CAM is blocked by the CFQ-specific flow control (Stop bit), if it is receiving control information ("Receiving Control" bit), and if the congestion information has been propagated to the output ports of the upstream switch ("Sent Stop" bit).

In Fig. 2 (pass 3), we show a "primary" CFQ allocation example. As can be seen, a new CFQ is dynamically allocated after the congestion detection at port P1 of switch 2. The associated CAM line is filled in with information about congestion: congested port identifier (P6 in this case), number of hops to reach that port (1 in this case), and the destinations of packets that request the congested port from P1 (at this first moment, only one destination: X).

[3] Note this is possible because we assume distributed deterministic routing; adaptive routing would not allow us to identify congested packets in that way.

From this moment, once a new CFQ is allocated in the input port, FBICM will separate the congested flows by storing in that CFQ those incoming packets whose destination is in the list-of-destinations field of the corresponding CAM line, while incoming packets belonging to non-congested flows will be mapped to the NFQ. Note that congested packets are identified taking into account only its destination, thus source (explicit) routing is not required. Congested packet identification is performed by the Packet Processing mechanism.

List-of-Destinations Implementation. Take into account that the list-of-destinations field described above is just a simplification (necessary for an straightforward understanding of FBICM basics), but it should be implemented efficiently in real systems, as it could become the bottleneck of the new logic or even consume many resources. One effective way to implement such list per CAM line is to add new fields to the routing table. Assuming a routing table with N entries (one per destination), a switch with p ports, and a FBICM mechanism with c CFQs per port, each entry for a destination node d would contain the following fields:

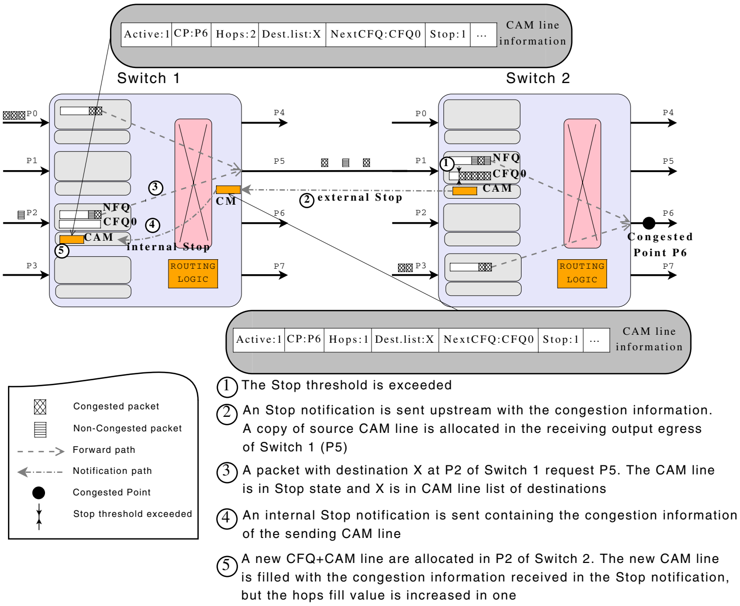
1. Output port (OP). This is the output port provided to the arbitration logic for the packet being routed, computed from packet destination.
2. CAMline at the output port (CLOP). A c-bit register indicating the CAM lines at the output port (OP) where the destination d is mapped into. Notice that the destination may be mapped on more than one CAM line, thus we need one bit per CAM line (c).
3. CAM line at the input port (CLIP). A vector with p elements, each with c bits. Each vector element represents the mappings of the destination d into the CAM lines on a particular port. Thus, each element contains c bits, one per CAM line.
4. Notification Sent bit (NSB). One bit per input port (p). Each bit will be set on each time a notification is sent upstream for the d destination through a given port.

As can be seen, several new fields are required at the routing table. Assuming a 1024-node system implemented with 8-port switches, a routing table with no congestion management would require 3 KB memory space (1024 destinations, each one coding a 3-bit output port). With the addition of the congestion management mechanism with 8 CFQs per port, the memory requirements for the routing table increase to 83 KB (1024 destinations, 3-bit for OP field, 4 bits for CLOP field, 32 bits for CLIP field, 8 bits for NSB field)[4].

3.4 Packet Processing

The post-processing mechanism decides if a packet belongs to a congested flow, thereby also deciding if it must be stored in a CFQ or in the NFQ. This mechanism looks around the packets arriving at the head of all the input port queues (NFQ and CFQs), and compares the packet destination to the CAM lines information (specifically, to the

[4] Although such memory increase may be acceptable, as future work we plan to reduce the memory requirements for FBICM. We plan to address caching mechanisms as destinations potentially will reach a switch through a small subset of ports.



1 The Stop threshold is exceeded

2 An Stop notification is sent upstream with the congestion information. A copy of source CAM line is allocated in the receiving output egress of Switch 1 (P5)

3 A packet with destination X at P2 of Switch 1 request P5. The CAM line is in Stop state and X is in CAM line list of destinations

4 An internal Stop notification is sent containing the congestion information of the sending CAM line

5 A new CFQ+CAM line are allocated in P2 of Switch 2. The new CAM line is filled with the congestion information received in the Stop notification, but the hops fill value is increased in one

Fig. 4. Operation Example of Congestion Information Propagation

destinations included in the list-of-destinations field). If the packet is at the head of the NFQ and the comparison result is a match, the packet will be considered as congested and will be moved to the corresponding CFQ in the same input port. Otherwise, the packet will remain in the NFQ and the post-processing mechanism will set it as ready for the switch arbiter. Note that this mechanism leaves at the head of the NFQ only non-congested packets, thus HOL blocking is avoided at the NFQ. On the other hand, once a packet stored in a CFQ reaches the head of a that queue, it will be set as ready for the arbiter. Note that this post-processing mechanism assures the in-order delivery of packets because, although congested and non-congested flows are separated, it keeps the order of packets belonging to the same flow (either congested or non-congested).

Additionally, the post-processing mechanism selects which NFQ or CFQ can forward a packet at each moment, by means of a round robin policy among all the queues (NFQ and active CFQs) at each input port.

3.5 Congestion Information Propagation

In the same way that FBICM sets a congestion detection threshold in the NFQ, a threshold (Stop threshold) is used, when congestion persists for long, both for avoiding CFQ overflow and for propagating congestion information. An example of this is shown in Fig. 4.

Basically, when the occupancy of a CFQ in an input port exceeds the Stop threshold, an Stop notification including the information of the associated CAM line is sent to the upstream output port. When an output port receives an external Stop notification, it checks if there is already an active CAM line containing the same received information. If not, it will allocate a new CAM line, filling it with that information (so, active output CAM lines will be exact copies of downstream CAM lines). Otherwise, if there were already an allocated CAM for the received information, the Stop notification is considered just as a flow control message. In both cases (new CAM line allocation or not), the involved CAM line will be set to Stop state, and as a consequence this output port will not accept packets belonging to packets flows whose destinations are included in the list-of-destinations field of that CAM line. Note that this means that the output port may control the flow of congested packets without requiring explicit routes for identifying them.

On the other hand, when the occupancy of an input port CFQ that sent an Stop notification decreases enough (until a given threshold), a Go notification is sent upstream, with the opposite effect than Stop ones. Thus, upon reception of a Go notification, an output port CAM line will turn off the Stop bit, entering the Go state and so unblocking the flow of packets.

Inside of the switch, “internal” Stop notifications may be sent when some input port request to forward a packet to an output port with active CAM lines. If one of these CAM lines is in Stop state and the packet destination is in its list-of-destinations field, an internal Stop notification, containing all the CAM line information, will effectively be sent to the input port. In this way, congestion information from downstream input port CAM lines are transmitted through output port CAM lines to all the input ports crossed by the flows contributing to congestion. Note again that specific routes are not necessary for identifying congested packets.

On its side, an input port receiving an internal Stop notification may allocate or not a new CFQ + CAM line, depending on the existence at that port of an active CAM line containing the same congestion information received. In the case of a new allocation, the CAM line will be filled with the received information, but increasing the “hops-to-reach” value by one (this reflects the growing distance to the “root of the congestion tree”). Otherwise, the internal Stop notification will be considered just as a flow control message. Again, in both cases (new allocation or just flow control), the involved CAM line will be set to Stop state. Of course, internal Go notifications are also sent for unblocking packet flow.

The newly allocated input port CFQ will store congested packets (again, identified only from their destinations), and may repeat the propagation process, spreading upstream the congestion information. In this way, when a congestion situation persists FBICM will identify the congested flows at any network point, and packets belonging to them will be stored in the corresponding CFQs. Thus, congested flows will not interact with the non-congested ones, thereby avoiding HOL blocking.

4 FBICM Evaluation

In this Section we will evaluate FBICM by comparing it to other HOL blocking elimination techniques, specifically to the RECN version for IQ switches (RECN-IQ)[6], Virtual Output Queues at network level (VOQNet) and Virtual Output Queues at switch level (VOQSw).

We will use different scenarios of traffic load and network size. For this purpose we have used an ad-hoc event-driven simulator developed in C language that models the network at the register transfer level. Firstly, we will describe the modeling assumptions and the main parameters used in the simulations. Secondly, we will analyze the obtained results.

4.1 Simulated Scenarios

The simulator models Bidirectional Multistage Interconnection Networks (BMINs) with switches, endnodes, and links, allowing different connectivity patterns (e.g. *perfect shuffle*, *butterfly*, ...) and network sizes. For this evaluation, we have used two netwok configurations:

1. Configuration 1: 64 hosts connected with a 64×64 *perfect shuffle* BMIN. This network includes 48 switches in three stages, each switch having 8 bidirectional ports.
2. Configuration 2: 256 hosts connected with a *perfect shuffle* 256×256 BMIN. This network includes 256 switches in four stages, each switch having 8 bidirectional ports.

In all the experiments, a BMIN self-routing deterministic algorithm has been used. For the RECN-IQ experiments, source routing has been modeled, thus this algorithm has been used to generate explicit routes included in packet headers. For FBICM, VOQNet and VOQSw, table-based, distributed routing has been modeled, thus in these cases the routing algorithm has been used for filling the routing tables.

At switches, packets are forwarded from input queues to output queues through a multiplexed crossbar, modeled with a speedup of 1 (link bandwidth is equal to crossbar bandwidth). RAM memories have been modeled at each input port of the switch, with different sizes depending on the simulated HOL blocking elimination technique. For FBICM, RECN-IQ and VOQSw, we have used 8 KB RAMs, but VOQNet requires larger memories since the number of queues per port is higher in this case. In particular, VOQNet requires 16 KB memories for 64×64 MINs and 64 KB memories for 256×256 MINs. In the case of FBICM, a maximum of 8 active CFQs per port have been allowed, and also 8 active SAQs the case of RECN-IQ.

In all the experiments, endnodes (hosts) are modeled as connected to switches using Input Adapters (IAs). Every IA is modeled with a fixed number of N admittance queues (where N is the total number of endnodes), and a variable number of injection queues, which follow a scheme similar to that of the output ports of a switch. When a message is generated, it is stored in the admittance queue assigned to its destination, and is packetized before being transferred to an injection queue. We have used 64-byte packets.

Of course, in order to compare the results of the considered HOL blocking elimination techniques, the simulator also allows experiments to be performed using either FBICM, RECN-IQ, VOQNet or VOQSw.

Regarding traffic load, the simulator allows to use either synthetic traffic or traces, and we have used both in our experiments. The considered synthetic traffic patterns are shown in table 1. Note that in some patterns (#1, #2 and #5) traffic follows a completely uniform (random) destination distribution, while in others (#3, #4 and #6), a percentage

Table 1. Synthetic traffic patterns used in the evaluation

		Random Traffic			Hot-Spot Traffic				
Traffic case	Network (BMIN)	# Srcs	Dest	Generation rate	# Srcs	Dest	Generation rate	Start time	End time
# 1	64×64	100%	random	100%	0%	-	-	-	-
# 2	64×64	100%	random	incremental	0%	-	-	-	-
# 3	64×64	75%	random	100%	25%	32	100%	1000 μs	1300 μs
# 4	64×64	75%	random	incremental	25%	32	incremental	0 μs	3000 μs
# 5	256×256	100%	random	100%	0%	-	-	-	-
# 6	256×256	75%	random	100%	25%	123	100%	1000 μs	1300 μs

of sources generate traffic addressed to a unique, hot-spot destination, thereby creating congestion. In all these patterns, packet generation rate is indicated as a relative percentage of link bandwidth.

Regarding traces, the ones used in our experiments were provided by Hewlett-Packard Labs, and they include all the I/O activity generated from 1/14/1999 to 2/28/1999 at the disk interface of the *cello* system. As these traces are almost nine years old, we have applied a time compression factor to the traces.

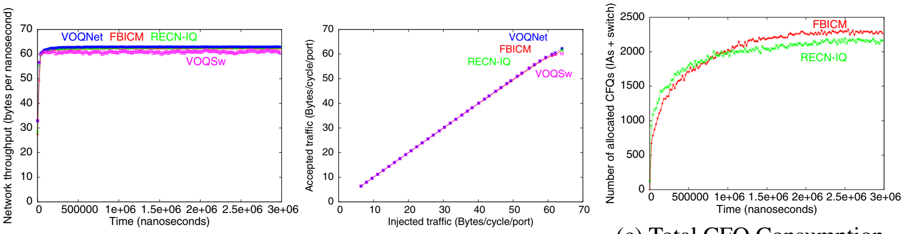
In all the experiments, the simulator offers different metrics. For our evaluation, we have considered network throughput (as a function of time or traffic load) as the main metric for measuring the performance of the networks when the different HOL blocking elimination techniques are used. Additionally, we have also considered FBICM consumption of CFQs (the total amount of active CFQs in the network at each moment) and RECN-IQ consumption of SAQs, in order to compare resource requirements of both techniques.

4.2 Evaluation Results

Figures 5(a) and 5(b) show network throughput results for completely uniform synthetic traffic patterns in a 64×64 BMIN. As we can see, FBICM obtains the same throughput as VOQNet and RECN-IQ, and it improves slightly the results of VOQSw. Due to the uniform traffic properties, the congestion situations are very short and occur in many different points in the network, thus none of the techniques faces great difficulties for keeping network performance at a good level.

However, for hot-spot scenarios leading to congestion situations in the same network, results are significantly different, as can be seen in Fig. 6(a) and 6(b). In these cases, network throughput significantly decreases when VOQSw are used, while FBICM, RECN-IQ and VOQNet achieve better performance, keeping network throughput at maximum even during congestion. Note, however, that VOQNet achieves these results by using far more queues per input port (specifically, 64 queues) than FBICM (1 queue + a maximum of 8 queues) or RECN-IQ (also 1 queue + a maximum of 8 queues). Since results obtained by these three techniques are quite similar, we can conclude that both FBICM and RECN-IQ eliminate HOL blocking more efficiently than VOQNet. But note also that FBICM achieves these results for distributed deterministic routing interconnects, while RECN-IQ cannot be implemented in such technologies.

On the other hand, Figs. 5(c) and 6(c) compare the total consumption of CFQs (FBICM) and SAQs (RECN-IQ) along simulation time when traffic cases #1 and #3 are used. As can be seen, results are very similar, thus we can conclude that FBICM

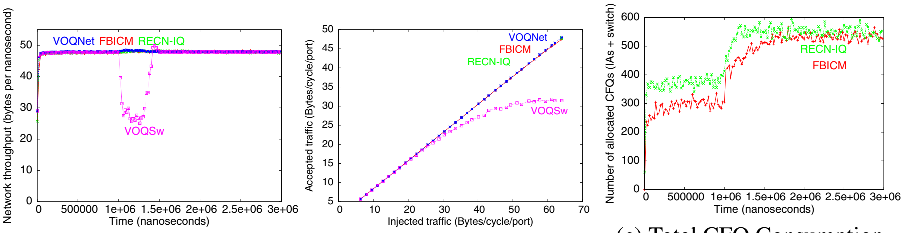


(a) Throughput vs. Time

(b) Throughput vs. Load

(c) Total CFQ Consumption vs. Time

Fig. 5. Network Throughput vs. Time and Load, and Total CFQ Consumption for Configuration 1 (Traffic Cases #1 and #2)



(a) Throughput vs. Time

(b) Throughput vs. Load

(c) Total CFQ Consumption vs. Time

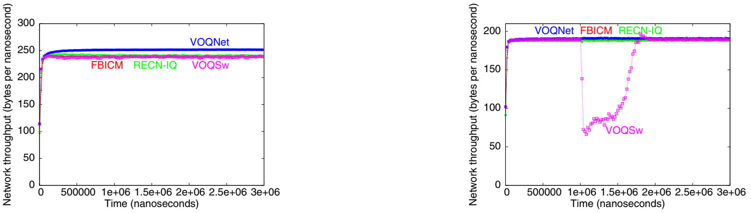
Fig. 6. Network Throughput vs. Time and Load, and Total CFQ Consumption for Configuration 1 (Traffic Cases #3 and #4)

require approximately the same amount of resources for allocating congested flows than RECN-IQ, thereby the former being approximately as efficient as the latter.

Figure 7 shows throughput results for a larger network (specifically, a 256×256 BMIN) when either completely uniform traffic (Fig. 7(a)) or hot-spot traffic (Fig. 7(b)) are present. In the uniform traffic case, FBICM achieves the same results as RECN-IQ with the same (and moderate) resources, while VOQNet achieves slightly better results. Note, however, that in this case VOQNet uses 64 KB memories per port for implementing 256 queues per port, while 8 KB per port are used by FBICM. As the congestion situations are very short in this traffic pattern, VOQSw obtains almost the same results as the other techniques. However, in the hot-spot case, the behavior of VOQSw is very poor, while we can see that FBICM achieved throughput is the same than VOQNet (but again the former using fewer queues than the latter). From this results we can conclude that also for large networks, FBICM keeps network performance at maximum even during congestion situations, thereby being a scalable technique.

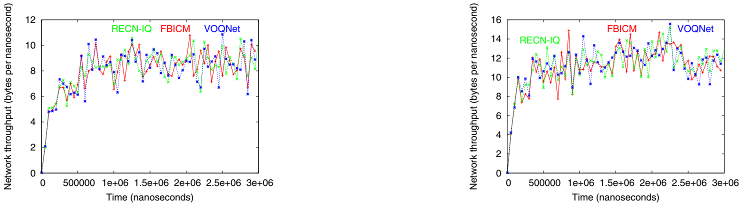
In the last figure of the evaluation (Fig. 8) it can be seen the similar throughput achieved by VOQNet, RECN-IQ and FBICM, when traces are used as traffic load in a 64×64 BMIN. It can be observed that, regardless of the traces compression factor, FBICM achieves the maximum throughput also for real traffic.

Summing up, all the evaluation results show that maximum performance is achieved by FBICM regardless of network size or traffic pattern, even in congestion situations. VOQNet obtains similar results than FBICM, but requires far more resources, especially



(a) Throughput vs. Time (Traffic Case #5) (b) Throughput vs. Time (Traffic Case #6)

Fig. 7. Network Throughput vs. Time for Configuration 2



(a) Compression Factor Set to 20 (b) Compression Factor Set to 40

Fig. 8. Network Throughput for Traces in a 64x64 BMIN

in large networks. VOQSw consumes similar resources than FBICM, but its performance is poor in hot-spot traffic situations, regardless of network size. On the other hand, results for RECN-IQ are similar to the FBICM ones, but in a different interconnect context.

5 Conclusions

Congestion situations are a serious menace for the performance of current interconnection networks. In these situations, packet flows contributing to congestion slow the advance of other flows due to the HOL blocking effect, thereby degrading overall network performance. Many techniques have been proposed for solving this problem, the one known as RECN achieving the best performance by eliminating HOL blocking in a truly efficient and scalable way. Since RECN requires the use of deterministic source routing, networks using table-based, distributed routing cannot benefit from a RECN implementation.

In this paper we have proposed FBICM, a new HOL blocking elimination technique for network technologies that implement table-based, distributed deterministic routing. Especially, we have described how implicit information about congested points can be stored and propagated (once congestion is detected) without requiring explicit routes, and how packets belonging to congested flows can be identified taking into account just their destination.

We have also shown evaluation results that demonstrate that FBICM achieves for distributed routing networks the same performance reached by the latest RECN version

(RECN-IQ) for source routing networks, since both techniques keep network performance at maximum even in serious congestion situations, regardless network size. Results also show that FBICM is more efficient than VOQnet (because the latter requires far more resources) and VOQSw (that exhibits a poor performance in congestion situations). Taking all this into account, we can conclude that FBICM offers efficient and scalable HOL blocking elimination to networks based on distributed deterministic routing networks.

Acknowledgments

This work is jointly supported by the following projects: Consolider Ingenio-2010-CSD2006-00046, TIN2006-15516-C04-02, PCC08-0078 (with PhD. grant A08/078).

References

1. Top-500-List: Web page (July 2008), `http://www.top500.org`
2. García, P.J., Flich, J., Duato, J., Johnson, I., Quiles, F.J., Naven, F.: Dynamic evolution of congestion trees: Analysis and impact on switch architecture. In: Conte, T., Navarro, N., Hwu, W.-m.W., Valero, M., Ungerer, T. (eds.) HiPEAC 2005. LNCS, vol. 3793, pp. 266–285. Springer, Heidelberg (2005)
3. Karol, M.J., Hluchyj, M.G., Morgan, S.P.: Input versus output queueing on a space-division packet switch. IEEE Trans. on Commun. COM-35, 1347–1356 (1987)
4. Duato, J., Johnson, I., Flich, J., Naven, F., García, P.J., Nachiondo, T.: A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks. In: Proceedings of the 11th Symposium on High Performance Computer Architecture (HPCA) (2005)
5. García, P.J., Flich, J., Duato, J., Johnson, I., Quiles, F.J., Naven, F.: Efficient, scalable congestion management for interconnection networks. IEEE Micro 26(5), 52–66 (2006)
6. Mora, G., García, P.J., Flich, J., Duato, J.: RECN-IQ: A cost-effective input-queued switch architecture with congestion management. In: Proceedings of 36th International Conference on Parallel Processing (ICPP 2007) (2007)
7. InfiniBand Trade Association: InfiniBand architecture specification volume 1. Release 1.0 (October 2000)
8. Wang, M., Siegel, H.J., Nichols, M.A., Abraham, S.: Using a multipath network for reducing the effects of hot spots. IEEE Transactions on Parallel and Distributed Systems 6(3), 252–268 (1995)
9. Thottetodi, M., Lebeck, A., Mukherjee, S.: Self-tuned congestion control for multiprocessor networks. In: Proc. of 7th. Int. Symp. on High Performance Computer Architecture (February 2001)
10. Dally, W., Carvey, P., Dennison, L.: Architecture of the Avici terabit switch/router. In: Proceedings of the 6th Symposium on Hot Interconnects, pp. 41–50 (1998)
11. Tamir, Y., Frazier, G.: Dynamically-allocated multi-queue buffers for vlsi communication switches. IEEE Transactions on Computers 41(6) (June 1992)
12. Smai, A., Thorelli, L.: Global reactive congestion control in multicomputer networks. In: Proc. 5th Int. Conference on High Performance Computing (1998)
13. Anderson, T., Owicki, S., Saxe, J., Thacker, C.: High-speed switch scheduling for local-area networks. ACM Transactions on Computer Systems 11(4), 319–352 (1993)

Achieving 10Gbps Network Processing: Are We There Yet?

Priya Govindarajan, Srihari Makineni, Donald Newell, Ravi Iyer,
Ram Huggahalli, and Amit Kumar

Intel Corporation, 2111 NE 25th Ave.,
Hillsboro 97124, USA
{priya.govindarajan,srihari.makineni,donald.newell,
ravishankar.iyer,ram.huggahalli,amit.kumar}@intel.com

Abstract. Scaling TCP/IP receive side processing to 10Gbps speeds on commercial server platforms has been a major challenge. This led to the development of two key techniques: Large Receive Offload (LRO) and Direct Cache Access (DCA). Only recently, systems supporting these two techniques have become available. So, we want to evaluate these two techniques using 10Gigabit NICs to find out if we can finally get 10Gbps rates. We evaluate these two techniques in detail to understand performance benefit offered by these two techniques and the remaining major overheads. Our measurements showed that LRO and DCA together improve TCP/IP receive performance by more than 50% over the base case (no LRO and DCA). These two techniques combined with the improvements in the CPU architecture and the rest of the platform over the last 3-4 years have more than doubled the TCP/IP receive processing throughput to 7Gbps. Our detailed architectural characterization of TCP/IP processing, with these two features enabled, has revealed that buffer management and copy operations still take up significant amount of processing time. We also analyze the scaling behavior of TCP/IP to figure out how multi-core architectures improve network processing. This part of our analysis has highlighted some limiting factors that need to be addressed to achieve scaling beyond 10Gbps.

Keywords: Large Receive Offload, LRO, Direct Cache Access, DCA, TOE, TCP/IP acceleration, de-fragmentation, receive offload, receive side coalescing, RSC.

1 Introduction

Scaling TCP/IP [17, 18] receive side processing to 10Gbps speeds is known to be a major challenge [11, 12, 13, 14, 15, 16]. There has been a significant effort in the industry and in academia to figure out ways of speeding up this processing. This has led to the development of a set of ideas and solutions ranging from integrating NIC devices tightly with the CPUs [23], to offloading the entire TCP/IP processing to NIC devices, usually referred as TCP/IP Offload Devices (TOE) [1,2,3,4], to some simple platform and stack optimizations that target specific overheads involved in the receive side processing. While integrating NIC devices with the CPU complex still remains largely a research activity, 10Gbps TOE devices are just starting to appear in the

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 518–528, 2008.

market. But these TOE devices are currently expensive and require special Linux OS as the mainstream Linux kernel does not support offloading the TCP/IP processing. So, our focus in this paper is on platform and stack optimizations that have been proposed to speed up the receive processing. The two main innovations in this category are Large Receive Offload (LRO) [6, 24] and Direct Cache Access (DCA) [8, 9]. These two techniques try to address two major sources of overheads in the TCP/IP receive processing. These are: 1) compulsory cache misses that happen while processing incoming packets and this includes the copy of data from kernel buffer to user buffer, and 2) per packet overheads such as protocol processing, buffer management, etc. Systems and NICs supporting these techniques have only recently started to appear in the market. This makes it possible for the first time to evaluate these techniques on real systems at 10Gbps speeds and to understand the benefit offered.

Our contributions in this paper are: a) detailed performance benefit analysis of LRO and DCA techniques, b) characterization of TCP/IP receive processing to understand remaining major overheads and impact of this processing on the processor architecture, so the industry can focus on eliminating these overheads to further improve the receive processing, and c) virtualization technology leads to system consolidation which in turn increases the demand on servers to support network bandwidths beyond 10Gbps. So, we wanted to understand how the TCP/IP processing scales with multi-core architectures. We have conducted various scaling experiments and highlight some key limiting factors to TCP/IP receive processing scalability. Our emphasis in this paper is on bulk data processing hence we focus only on large transfer sizes (>2KB). Our measurements showed that these two techniques together improve TCP/IP receive performance by more than 50%.

The rest of the paper is organized as follows. We start out with a brief overview of TCP/IP receive side processing in section 2. We then describe the two key optimizations to TCP/IP receive side processing, LRO and DCA, in section 3. In section 4, we explain the measurements methodology, tools used, system setup and tested configurations. In section 5, we present measurements data, analysis of benefits offered by LRO and DCA and scaling characteristics. We conclude the paper with Summary and Conclusions in section 6.

2 TCP/IP Receive Processing

In this section we provide a high level overview of the processing that takes place from the time a NIC receives an Ethernet frame till the incoming data is handed over to the intended application. It is not our intention to provide a detailed description of this processing, but to provide sufficient context for readers while highlighting some major overheads involved in this processing.

Receive-side processing begins when NIC hardware receives an Ethernet frame from network. NIC extracts Ethernet frame delineation bits and CRC value and validates the frame. Today's NICs also perform checksum computations for the TCP and IP portions of the packet and compare those with checksum values TCP and IP headers. In order to notify the software stack about incoming packets and their placement in the memory, NIC uses descriptors that are arranged in a circular ring fashion. Descriptor data structure is typically 16bytes and contains among other things, address of a memory buffer (NIC buffer) to store the incoming packet data. NIC copies the incoming data at the memory location specified in the descriptor using onboard DMA engine. Once the packet is placed in memory, NIC updates a status field inside the descriptor to indicate to the driver that this descriptor holds a valid packet and generates an interrupt. This kicks off the SW processing of the received packet.

Figure 1 shows TCP/IP receive processing flow. The Ethernet device driver reads the descriptor and makes sure that NIC has indicated that this is a valid packet. Driver then classifies the packet as either IP packet or some other. If it is an IP packet then it forwards it to the TCP/IP stack for further processing. Since this descriptor was updated by NIC earlier, it results in invalidating processor's copy (if found in cache). So the processor would have to fetch the descriptor from the main memory. If the descriptor size is 16 bytes then each cache line (64 bytes) can accommodate up to 4 descriptors. Similarly, accessing packet headers by TCP/IP stack also results in cache misses as this data was just placed in the memory by the NIC. TCP/IP headers combined, without any option fields, is 40 bytes long. So each packet header will result in one compulsory cache miss. The next step in processing is to identify the connection to which this packet belongs. TCP/IP software stores state information of each open connection in a data structure, called the TCP/IP Control Block (TCB). Since there can potentially be several thousand open connections, hence many TCBs, TCP/IP software uses a well known search mechanism called hashing for fast lookup of the right TCB. The hash value is calculated by using the IP address and port number of both the source and destination machines. Several fields (sequence numbers for received/acknowledged bytes, application's pre-posted buffers, etc.) inside the TCB are updated whenever a new packet is received. TCP/IP stack then needs to figure out where to copy the packet payload (data portion). It checks to see if the target application has already posted a buffer to receive incoming data. If a buffer is available, stack copies the data from the NIC buffer into that buffer. Otherwise, it will wait for the application to provide a buffer. TCP/IP stack may be forced to copy the data into a temporary buffer if application didn't provide one. When the incoming data is first copied, source buffer results in compulsory cache misses as the data has to be read from main memory.

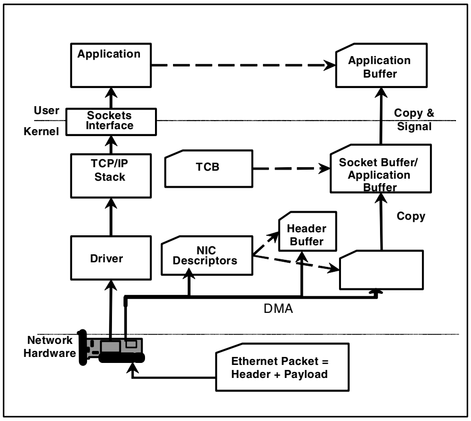


Fig. 1. Data Flow in Receive-Side processing

3 New Techniques to Accelerate TCP/IP Receive Processing

In this section we cover LRO [6,24] and DCA [8,9] techniques and explain how these techniques work and address major overheads involved in TCP/IP receive side processing. Our intention is not to cover every detail about these two techniques as these have been already covered elsewhere.

3.1 Large Receive Offload (LRO)

The main goal of LRO is to reduce per packet TCP/IP processing costs which are significant. These per packet processing costs include not only protocol processing cost, but all the other associated costs around buffer management, interfacing with the application, etc. LRO accomplishes this by passing fewer but larger packets to the TCP/IP stack for processing. LRO is a software mechanism that is typically implemented in the NIC driver. LRO identifies incoming packets that belong to same TCP/IP connection and coalesces these packets into a single larger packet. LRO can simultaneously coalesce packets for several connections. The coalesced packet has a single TCP/IP header that represents the entire coalesced packet. Coalescing TCP/IP packets is possible because TCP is a byte stream protocol and applications can't make any assumptions about boundaries of a message. For example, an application can receive tail end of a message and the beginning of the next message in the same 'receive' call.

LRO works in conjunction with another receive side optimization, called interrupt moderation. Interrupt moderation forces the NIC device to interrupt the CPU once after it receives some pre configured number of packets or certain time has elapsed since the last interrupt. This mechanism reduces the number of times the NIC interrupts the CPU. So, during each interrupt NIC may have received some number of packets (~ 50-100 packets per interrupt) that need to be processed. This allows the LRO code to sort through these packets and coalesce packets that belong to same connections. At the end of each interrupt processing, LRO stops coalescing and the NIC driver sends these coalesced packets to the TCP/IP stack for further processing. LRO code applies certain criteria to decide whether to coalesce a packet or not. It makes sure that the incoming packet is a valid TCP/IP packet, has correct sequence number, etc. If the incoming packet has TCP PUSH bit set, then LRO might decide to stop coalescing and send the coalesced packet up the stack for further processing. Before sending the coalesced packet up, the LRO code attaches a new TCP/IP header to the coalesced packet indicating the correct sequence number and size among other things. During packet coalescing, payload data in the incoming packets is not copied into a larger buffer, but the payload buffers are chained together in a queue. LRO code maintains separate queues, one for each TCP/IP connection. At the end of each interrupt processing, all the queues are flushed meaning that all the coalesced packets are sent up and coalescing starts fresh when the next interrupt comes.

This LRO functionality can be offloaded to the NIC device as was proposed by Makineni, et al. in [6]. This further maximizes the gains as the CPU does not have to run the coalescing code. Coalescing in software is about 100 instructions/pkt and on average takes 400 to 500 cycles. They have also shown that TCP/IP traffic is bursty [5,7] in nature and packets that belong to the same connection come in quick succession if not back to back even when there are 100s of active connections.

3.2 Direct Cache Access (DCA)

In section 2, we have pointed out how receive processing suffers from compulsory cache misses when fetching new descriptors, TCP/IP headers and payload data. DCA [8,9] aims to eliminate these compulsory cache misses by prefetching this data into the processors cache in advance. When the NIC updates the descriptors or writes header and payload data to system memory using DMA operation, snoops are sent to the processor to invalidate the processor copy of these lines. DCA listens to these snoops and starts prefetching these lines. As a result, descriptors that are pointing to the new packets and incoming packet headers and payload data are brought into processor's cache ahead of time. When the processor is ready to process the new packets, it will find the data in the cache hence does not have to go to memory. Memory accesses are expensive and can take up about 20-25% of total CPU time per packet. Eliminating at least some of these cache misses should greatly improve the receive processing. Hardware prefetchers try to do the same by prefetching the data into the cache, but they only get triggered when the processor starts accessing the adjacent lines. Hence they will not be able to hide the entire memory access latency. In addition, these prefetchers typically result in wasting precious memory bandwidth by bringing in more data than what is needed. Our measurements show that even though the prefetchers in Intel® Core 2™ Duo line of processors are very efficient, DCA still has significant advantage, especially when combined with the LRO. DCA has two benefits: 1) timely availability of data in cache leading directly to a lower average memory latency and 2) reduction in memory bandwidth requirement. An ideal implementation of DCA would eliminate the need to write data to memory, continuously updating cache lines in the cache with new data.

4 Measurements Methodology and Setup

In this section, we describe our test setup, different configurations for which measurements data has been collected and various tools used to measure TCP/IP performance and to profile the processing.

4.1 System Setup

Our System Under Test (SUT), receiver machine, is an MP system equipped with X7350 Quad-Core Intel® Xeon® Processors running at 2.93GHz frequency. It has Intel 7300 server chipset with data traffic optimization and supports 1066MTS Front Side Bus (FSB). Each processor has 2 last level caches of 4MB each that are shared by 2 cores. We have removed three processors from the system for our measurements and analysis purposes. Our client system is a Dell PowerEdge server with an Intel Xeon Quad Core 2.99Ghz processor and 24G of memory. Our client system is powerful enough that it never became a bottleneck during our testing. We have used a PCI Express based single port 10Gbps NIC from Intel (EXPX9501 series) in both the receiver and sender machines. The 7300 chipset supports DCA. The SUT was running Linux kernel 2.6.22.9 and ixgbe NIC driver version 1.1.21 and ioatdma module version 1.23. We have connected the SUT and the client back to back without going through any switch or router. We have used TCP bandwidth measurement application called IPerf to generate traffic between the sender and the receiver.

4.2 Configurations Tested

A number of features (DCA, LRO and prefetchers) were evaluated to understand their individual impact on performance as well as their combined impact. Four cases – Base, DCA, LRO, DCA+LRO are evaluated both with and without prefetchers. For each of the above 8 cases, we evaluated the performance for different packet sizes ranging from 64B to 64KB but only behavior of packet sizes greater than 2K is discussed here. A primary goal of the analysis was to understand how the performance of TCP/IP receive processing scales with more cores and we therefore looked at the performance with 1 core and then 2 cores.

4.3 Tools

IPerf
IPerf [20] is a network bandwidth measurement tool that is publicly available. IPerf runs in client and server modes and transfers data between them. We have modified the public version, because we have observed, during our 10GbE experiments, that the worker threads (that transmit or receive) were gated by a reporting thread (that keeps track of traffic statistics). We modified Iperf's reporting structure and ensured that worker threads are always free running to achieve maximum performance.

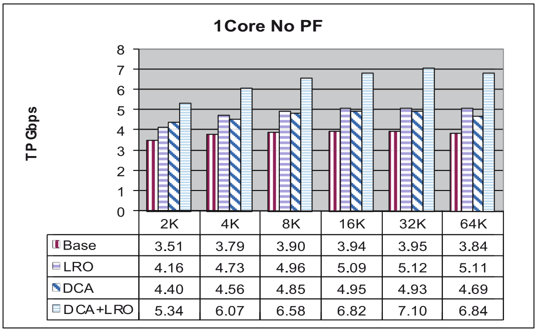
NTttcp
NTttcp [20] is the Microsoft Windows version of the popular network testing tool, ttcp [19]. This is a multithreaded, asynchronous application that sends and receives data between two or more end points. This tool measures network performance in terms of network bytes transferred per second and the CPU cycles per byte. This has two executables one for the client and the second for the server.

5 Results and Analysis

In this section, we show our measurements data and discuss the benefits of LRO and DCA. We show performance monitoring data collected from the platform to explain where the benefits are coming from and what the remaining overheads are. We start out with measurements data collected using one core and then move to multi core scenarios.

5.1 TCP/IP Performance on Single Core

Figure 2 shows a comparison of TCP/IP receive side performance for four different combinations. These are denoted as Base, LRO, DCA and DCA+LRO in the graph. For these measurements, we have turned off hardware prefetchers on the platform. The x-axis in the graph shows application payload sizes transferred in KB and the y-axis shows total throughput achieved in Gbps. CPU utilization is 100% for all the



	2K	4K	8K	16K	32K	64K
Base	3.51	3.79	3.90	3.94	3.95	3.84
LRO	4.16	4.73	4.96	5.09	5.12	5.11
DCA	4.40	4.56	4.85	4.95	4.93	4.69
DCA+LRO	5.34	6.07	6.58	6.82	7.10	6.84

Fig. 2. TCP/IP Performance on 1 Core without hardware prefetchers

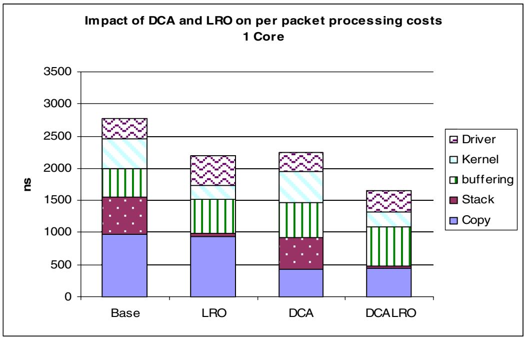


Fig. 3. Functional level breakdown of per packet processing cost

tests. For 8KB transfers, the Base case achieves about 3.9Gbps throughput. Roughly comparing with published numbers from 3-4 years ago, this is about 1Gbps higher. So, this improved performance can be attributed to improvements in the processor architecture, increased frequency (only slightly) and increased memory and system interconnect speeds.

LRO and DCA at 8KB payload size offer about 1Gbps higher throughput, which is about 27% over the Base case. However, the combined benefit of LRO and DCA is about 68% which is significantly higher than the sum of individual benefits. LRO reduces overall instructions executed per packet from ~2200 in Base case to 1350 – a saving of 850 instructions.

In Figure 3, we show break down of total time spent per packet on average for all four different configurations. We have collected this data using SEP tool. SEP is an internal tool that is somewhat similar to Linux tool, called OProfile [27]. SEP tool gave time spent by the core in each function during TCP/IP processing. We have categorized these functions into higher level components for ease of analysis and understanding.

This data shows that LRO compared to the Base case reduces time spent in stack processing (578 to 41ns) and kernel (463 to 216ns), but increases time spent in the Ethernet driver (304 to 453ns) and buffer management (454 to 537ns). Since the LRO code runs in the driver, it explains why the driver code is taking more time over the Base case. Interesting point here is that LRO not only reduces per packet processing costs but also significantly reduces time spent in the kernel component. Overall, LRO saves about 600ns per packet. DCA saves on average about 500ns per packet compared to the base and most of the savings is in the copy function (972 to 431ns). However, LRO+DCA together save on average about 1100ns per packet over the base case. Looking at the time breakdown for this case reveals that copy and buffering functions still take significant amount of time hence should be ideal candidates for further optimizations.

Next, we wanted to figure out how hardware prefetchers on this platform would perform relative, and whether they would offer same level of benefit as DCA. We found out that the prefetchers offer roughly the same level of benefit as DCA. However, if we look at LRO with prefetchers the throughput is in general lower than LRO+DCA without prefetchers. The main reason for this is that the prefetchers bring in lot of unnecessary data wasting bus and memory bandwidth thus increasing the memory access latency.

5.2 TCP/IP Performance on Two Cores Sharing the Last Level Cache

In this section, we discuss the impact of LRO and DCA on TCP/IP performance when two cores sharing the same last level cache are used. The two cores chosen share the same last level cache and we have directed all the network traffic to these two cores by affinitizing the IPerf program and the interrupts from the NIC. NIC interrupts are affinitized using smp_affinity of MSI queues. We made sure that the other two cores were completely idle during the measurements. The graph in figure 4 shows this data for the same 4 configurations described above. The graph also shows CPU utilizations for LRO and DCA+LRO configurations on the secondary y-axis. CPU utilization was 100% for the other configurations. Throughput numbers have gone up in general compared to 1 core case. We discuss more about scaling behavior later in this paper.

DCA benefit over the Base is only about 300 Mbps vs. 1Gbps in the 1 core case. We have noticed that DCA reduced 1 less cache miss than in the 1 core case. Also, we have noticed that DCA has increased memory access latency by about 12% over the Base case. We suspect that some of the lines that the DCA has brought into the cache were getting kicked out before they get used due to higher lag time and due to 2 cores sharing the same 4MB cache. We are still trying to root cause this.

LRO has offered either CPU savings or higher throughput compared to the base case. Reason for lower throughput and lower CPU utilization as in 2KB payload size is because we were trying to coalesce up to 44 packets at once and this was resulting in packets not getting processed by the upper layers in a timely manner. When we changed the degree of coalescing to 20 packet max, then we were able to achieve higher throughput numbers and higher CPU utilizations. More experiments are needed to determine the correct number that works across all payload sizes. For 8KB payload size, both LRO and LRO+DCA offer significant advantage over the base case. LRO+DCA achieves line rate (9.3gbps) at 85% cpu utilization (170% if we add up utilization of 2 cores). Base case achieved only 7Gbps. If we were to extrapolate the LRO+DCA throughput for 100% utilization then we get about 60% higher throughput than the Base case.

We ran the same experiments with prefetchers turned on and we have observed similar behavior as in 1 core case, so we are not showing this data here due to limited space.

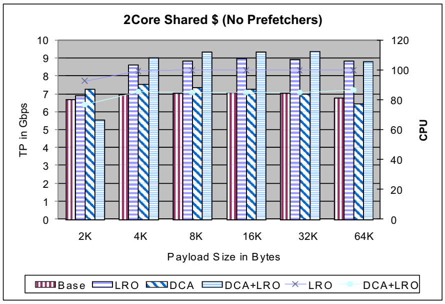


Fig. 4. TCP/IP Performance on 2 Cores sharing cache

5.3 Two Core TCP/IP Performance with Two Different Last Level Caches

To understand the effects of multiple last level caches on TCP/IP processing we have measured performance numbers using 2 cores that are mapped to 2 different last level caches. Both the cores are still in the same CPU. This scenario is similar to what happens when multiple sockets are employed to do network processing. Generally speaking, even though the performance is higher than that of 1 core case, it is lower than the 2 core shared cache case. If we extrapolate LRO+DCA numbers to 100% CPU utilization, then we see about 20% benefit over the Base. DCA performance was down compared to the base case as a result of higher cache misses. This is because there is no synchronization between where the application (IPerf) would run and consume the data and which last level cache that DCA is prefetching data into. We will discuss this issue more in the later section. DCA case has 2 times more cache misses compared to shared cache scenario.

5.4 Analysis of Scaling Behavior

In this section, we analyze the scaling behavior of TCP/IP processing when multiple cores and caches are employed. We focus on LRO and DCA scalability only as the other aspects of scalability have been already investigated [25, 26]. Table 1 below shows how DCA benefit scales going from 1 core to 2 cores with shared cache. DCA benefit in 2 cores shared cache case has reduced significantly. We believe that this is because of cache pollution caused by prefetching data well in advance in time. Cache pollution was not a problem in 1 core case because it has all 4MB cache to itself, but each core logically has only 2MB cache when both the cores are enabled.

Table 1. Percentage increase due to DCA over base (DCA- Base)/ Base

Payload Size in Bytes	1 Core	2 Core Shared $
4K	21%	11%
8K	24%	5%
16K	26%	3%
32K	25%	0%
64K	22%	-4%

In case of separate caches, we ran into a different issue that affected DCA benefit scaling negatively. It has to do with which cache DCA is prefetching data into. We found out that about 50% of time the data is being prefetched into the wrong cache limiting DCA benefit scalability. Aligning interrupts, interrupt processing and packet processing and where the copy operation happens all affect scaling. Currently, there is no easy way to align all these things so they all happen on the same core in Linux. To achieve perfect scalability in this situation, one has to use multiple NICs (or, multi ported NIC) and run multiple instances of IPerf where each IPerf is affinitized to the same core/cores as the interrupts are.

On the other hand, we saw LRO scaling reasonably well from 1 core to 2 cores shared cache. LRO scaling from 1 core to 2 cores with split cache has been affected the same way as the Base case. Buffer management costs have almost doubled going from 1 core to 2 core split cache scenario impacting overall scalability.

6 Summary and Conclusions

In this paper, we have evaluated two recently proposed techniques, namely LRO and DCA to accelerate receive side TCP/IP processing using latest server systems. Our measurements showed that while these two techniques have improved TCP/IP processing efficiency significantly a single CPU core still can't achieve 10Gbps rates when receiving and processing bulk data (>2KB). For 8KB size transfers, LRO and DCA together offer bout 70% higher throughput over the base case.

We have collected performance monitoring information and functional level breakdown of processing time to understand how LRO and DCA change the processing times and to highlight what the remaining major overheads are. Data copy and buffering costs are still higher, and lowering these overheads is critical to achieving 10Gbps rates. Our analysis of TCP/IP receive side processing scaling showed that the processing with LRO+DCA scales well (67% for 8KB) when multiple cores sharing the last level cache, but it drops to 15% when cores don't share the same last level cache. The main reason for poor scaling in case of multiple caches is due to DCA prefetch hints going to just one core. We are currently investigating this issue.

As for our future work, we would like to evaluate offloading LRO code on to the NIC [5] and the use of data copy engines [23] to reduce copy overhead. We want to investigate scaling issues and investigate potential solutions.

References

1. Alacritech SLIC: A Data Path TCP Offload methodology, `http://www.alacritech.com/html/techreview.html`
2. Mogul, J.: TCP offload is a dumb idea whose time has come. In: A Symposium on Hot Operating Systems (HOT OS) (2003)
3. Rangarajan, M., et al.: TCP Servers: Offloading TCP/IP Processing in Internet Servers. Design, Implementation, and Performance, Rutgers University, Technical Report, DCS-TR-481 (March 2002)
4. Regnier, G., Makineni, S., Illikkal, R., Iyer, R., et al.: TCP onloading for data center servers. IEEE Computer 37(11), 48–58 (2004)
5. Blanton, E., Allman, M.: On the Impact of Bursting on TCP Performance. In: Proceedings of the Workshop for Passive and Active Measurement (March 2005)

6. Makineni, S., Iyer, R., Sarangam, P., Newell, D., Zhao, L., Illikkal, R., Moses, J.: Receive Side Coalescing for Accelerating TCP/IP Processing. In: Robert, Y., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2006. LNCS, vol. 4297, pp. 289–300. Springer, Heidelberg (2006)
7. Kurmann, C., Müller, M., Rauch, F., Stricker, T.M.: Speculative defragmentation— A technique to improve the communication software efficiency for gigabit Ethernet. In: Proc. 9th IEEE Symp. High Performance Distr. Comp., Pittsburgh (August 2000)
8. Huggahalli, R., Iyer, R., Tetrick, S.: Direct Cache Access for High Bandwidth Network I/O. In: 32nd Annual International Symposium on Computer Architecture (ISCA 2005) (June 2005)
9. Kumar, A., et al.: Impact of Cache Coherence Protocols on the Processing of Network Traffic. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40) (2007)
10. Chase, J., et al.: End System Optimizations for High-Speed TCP. In: IEEE Communications, Special Issue on High-Speed TCP (2000)
11. Clark, D.D., Romkey, J., Salwen, H.: An analysis of TCP processing overhead. IEEE Communications 27(6), 23–29 (1989)
12. Foong, A.P., Huff, T.R., Hum, H.H., Patwardhan, J.P., Regnier, G.J.: TCP Performance revisited. In: Proc. IEEE Int. Symp. on Performance of Systems and Software, Austin, pp. 70–79 (March 2003)
13. Makineni, S., Iyer, R.: Architectural Characterization of TCP/IP Packet Processing on the Pentium M microprocessor. In: Int'l. Conf. on High Performance Computer Architecture (HPCA-10) (February 2004)
14. Makineni, S., et al.: Measurement-based Analysis of TCP/IP Processing Requirements. HiPC Poster Presentation (2003)
15. Kay, J., Pasquale, J.: The importance of non-data touching processing overheads in TCP/IP. In: Proc. ACM SIGCOMM, San Francisco, pp. 259–268 (October 1993)
16. Mogul, J.C.: Observing TCP Dynamics in Real Networks. In: ACM SIGCOMM, pp. 305–317 (1992)
17. Postel, J. (ed.): Internet Protocol - DARPA Internet program protocol specification, RFC 791 (September 1981)
18. Postel, J.B.: Transmission Control Protocol, RFC 793, Information Sciences Institute (September 1981)
19. The TTTCP Benchmark, http://ftp.arl.mil/~mike/ttcp.html
20. NTttcp, http://www.microsoft.com/whdc/device/network/TCP_tool.mspx
21. http://dast.nlanr.net/Projects/Iperf
22. Zhao, L., et al.: Hardware Support for Bulk Data Movement in Server Platforms. In: Proceedings of ICCD 2005 (2005)
23. Binkert, N., et al.: Integrated network interfaces for high-bandwidth TCP/IP. In: Proceedings of the 2006 ASPLOS Conference (December 2006)
24. Grossman, L.: Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In: Ottawa Linux Symposium, Ottawa (2005)
25. Foong, A., Fung, J., Newell, D.: Improved Linux* SMP Scaling: User-directed Processor Affinity,
http://softwarecommunity.intel.com/articles/eng/1781.htm
26. Scalable Networking: Eliminating the Receive Processing Bottleneck – Introducing SS. Microsoft WinHEC (April 2004)
27. About OProfile, http://oprofile.sourceforge.net/about/

SAIL: Self-Adaptive File Reallocation on Hybrid Disk Arrays

Tao Xie and Deepthi Madathil

Department of Computer Science, San Diego State University,
San Diego, CA 92182, USA
xie@cs.sdsu.edu, madathil@rohan.sdsu.edu

Abstract. Flash-memory based solid state disks, though currently more expensive and inadequate in write cycles, offer much faster read accesses while consume much less energy compared with hard disk drives. In order to gain complementary merits of hard disks and flash disks, we propose a hybrid disk array based storage architecture for data-intensive server-class applications. Further, on top of the proposed storage architecture, a _s_elf-_a_daptive f_i_le rea_l_location strategy, called SAIL, which is able to adapt to dynamically changed file access patterns, is developed. Comprehensive trace-driven experiments demonstrate that compared with a very recent file placement technique PB-PDC, which also employs the combined advantages of a hard disk and a flash memory device, SAIL exhibits its strength in both performance and energy consumption while maintains the reliability of flash disks by confining their write cycles.

Keywords: File reallocation, flash disk, hybrid disk array, energy conservation.

1 Introduction

File assignment problem (FAP), the problem of allocating a set of files onto a disk array before they are accessed so that some cost functions or performance metrics can be optimized, have been extensively studied [9][16][20][22][23][24]. Typically, file assignment algorithms reported in the literature can be categorized into two camps: static [16][23][24] and dynamic [2][20][22]. While static file assignment algorithms require a prior knowledge about the workload statistics, dynamic file assignment strategies can adapt to varying access patterns without the prior information of the characteristics of files. In this paper, we address the problem of dynamically assigning and reallocating files in a hybrid disk array storage system where hard disks and flash disks are structured in a RAID-0 fashion, respectively.

1.1 File Allocation and Reallocation

There have been many studies [16][23][24] over static file assignment problem where the following two assumptions are held: (1) all files are to be allocated at the same time; (2) the access frequency of each file is known as a priori and it does not change over time. In reality, however, these two assumptions are largely unrealistic. This is

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 529–540, 2008.

because file systems are highly dynamic, which implies that many files are created or deleted on the fly [22]. Moreover, the access pattern of a file system might change over a long-term period [19]. Therefore, dynamic file allocation and reallocation algorithms, which are able to intelligently allocate files dynamically created and to reorganize files to adapt to varying access pattern become indispensable.

Unfortunately, compared with numerous static file assignment algorithms [16][23][24], only very few investigations on dynamic file allocation [22] and reallocation problem [20] have been accomplished. Weikum et al. first proposed an array of heuristic algorithms for the placement of dynamically created files on a hard disk array [22]. Later on they extended their algorithms to accommodate dynamic redistributions of the data when access patterns change [20]. However, all of their algorithms bear the following three major limitations [20]. First, they assume that all of the subrequests are uniformly distributed among the disks, which obviously contradicts the fact that real workloads generally exhibit skewed access frequencies [16][19]. Second, their approaches just assume that the relevant workload parameters can be estimated with sufficient accuracy without actually implementing any dynamic file access monitoring mechanisms. Finally, all their algorithms employ a file-specific striping policy, which means the size of a stripe unit is file-dependent. The non-uniform file striping method is not practical because it will impose a prohibitive overhead on disk array controller. Therefore, a new dynamic file allocation and reallocation strategy without the limitations mentioned above is needed to fully address the challenging dynamic file assignment and reorganization problem. Besides, the new strategy should be energy-aware as disk arrays contribute a significant percentage of total energy consumption in a computing infrastructure.

1.2 Why Flash Disks?

Flash memory is useful for more than just consumer devices. Current flash memory assisted hard disk storage systems are mainly proposed to be applied in mobile platforms like personal laptops [6][15] or embedded systems [4]. Essentially, these flash memory and hard disk mixed storage systems only take flash memory as an extra layer of cache buffer [1] [15]. Very recently, Kim et al. extended the usage of flash memory device by developing an energy-efficient file placement technique named PB-PDC (pattern-based PDC), which adapts the existing PDC (Popular Data Concentration) algorithm [17] by separating read and write I/O requests. More precisely, PB-PDC locates all read-only data on a flash drive while puts all the rest of data on a hard disk. Still, the PB-PDC technique only concentrated on one flash drive with a single hard disk in a mobile laptop computing environment.

We believe that the application of flash memory can go far beyond personal mobile computing and embedded system domains because it is also well-suited for enterprise level applications, where performance, energy conservation, and disk reliability need to be taken into account simultaneously [4]. Compared with hard disk drives, flash disks possess the following salient advantages [10][18]. First, they inherently consume much less energy than mechanical mechanism based hard disks [4]. Second, because of their solid state design they are free of mechanical movements, and thus, their reliability are enhanced. Third, they offer much faster random access without seek time delays and rotation latencies [13][14]. The main concern on current flash disks is their considerably higher prices. Therefore, it is wise to integrate small capacity flash disks with high capacity hard disk drives to form a hybrid disk array so that their complementary merits can be benefited by enterprise applications.

1.3 Self-Adaptive File Reallocation

In this section we re-examine the dynamic file allocation and reallocation problem in the context of a hybrid disk array. Flash disks, though energy-saving in nature, have inferior performance in write speed compared to hard disks. Besides, they have limited number of erasure cycles. To fully exploit the advantages of flash disks and hard disks, we develop a *s*elf-*a*daptive f*i*le rea*l*location strategy named SAIL. SAIL dynamically monitors the access patterns of each file. Files can be dynamically created or deleted. In addition, the access pattern of each file could vary over time. Initially, all files including newly created files are distributed across the hard disk array in RAID-0 manner. At the end of each epoch, after obtaining statistics of each file's access pattern, SAIL first separates all files into three broad categories: write-excessive, read-exclusive, and read-write. If the frequency of a file's write accesses exceeds the suggested flash disk write frequency threshold value (e.g., 1 million times within 5 years), it will be defined as a write-excessive file and will stay on the hard disk array. All rest files will be further divided into two groups: read-exclusive and read-write. Files with both read and write accesses are in the read-write group, whereas files with only read accesses go into the read-exclusive group. Next, SAIL selects a set of files that are appropriate for being allocated on the flash disk array from the read-exclusive and the read-write groups based on each file's popularity, performance gain pg_i (Eq. 4), and energy gain eg_i (Eq. 5). And then SAIL reallocates these files onto the flash disk array. When file access pattern changes, SAIL redistributes files between the flash disk array and the hard disk array accordingly.

Based on the observations from some real-life traces [8][19], the popularity of a piece of data normally does not change dramatically in a short period of time. Thus, we argue that although the access pattern of a particular file may noticeably vary over a long run it only smoothly changes within each epoch. Therefore, it is feasible for SAIL to use the most recent access statistics of a file to predict its next epoch file access pattern in a dynamic I/O workload scenario.

2 The Hybrid Storage Architecture

There are two main types of flash memory in the market: NAND flash memory and NOR flash memory [4]. Since NAND flash memory is more appropriate for data storage [4], we only consider NAND flash memory in this paper. Also, there are two options when one implements a flash memory based storage system: emulating a flash disk as a block-device like a hard disk or designing a brand new native file system directly over flash disks. We adopt the first approach as it introduces little change of an operating system running on a host machine. In order to integrate a flash disk into an existing storage system, two important layers of software modules that sit between the file system and the flash disk are indispensable [12]. They are MTD (memory technology device) driver and FTL (flash translation layer) driver. Lower-level functions of a storage medium such as read, write, and erase are provided by the MTD driver. Supported by the underlying lower-level functions offered by the MTD driver, the FTL driver implements higher-level algorithms like wear-leveling, garbage collection, and physical/logical address translation [12]. With the assistance of the FTL driver, the file system can access the flash disk as a hard disk without being aware of the existence of the flash disk. How to design and implement these two software layers of modules is out of the scope of this paper. We assume that MTD and FTL drivers exist between file system and the flash disk.

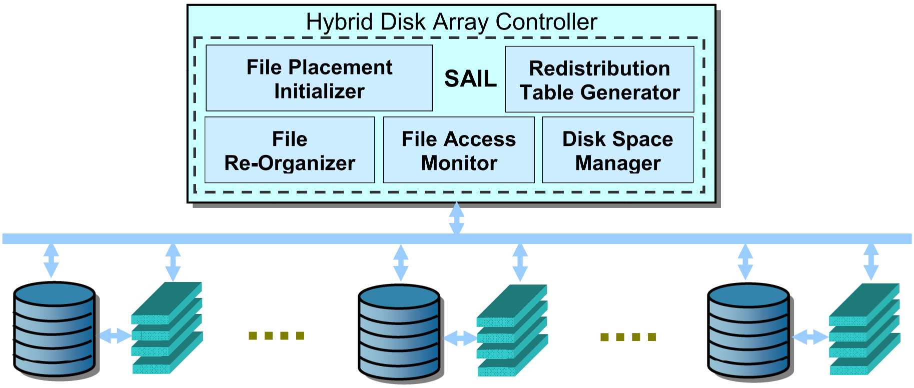

Hybrid Disk Array Controller
File Placement Initializer
SAIL
Redistribution Table Generator
File Re-Organizer
File Access Monitor
Disk Space Manager

Fig. 1. Overview of the hybrid disk array architecture

The hybrid disk array storage architecture is depicted in Fig. 1, where both hard disks and flash disks are directly attached to the system bus. All hard disks are organized in a RAID structure like RAID-0. Similarly, all flash disks are managed in the same RAID organization as the hard disk array. Besides, the number of hard disks is equal to the number of flash disks and each flash disk cooperates with a hard disk through a dedicated high-bandwidth connection to compose a disk pair. The rationale behind the disk pair configuration is three-fold. First, the equal number of the two types of disks makes balancing load between the hard disk array and the flash disk array easier. Second, it simplifies file reallocation between the two disk arrays. Last but not least, the disk pair configuration obviously enhances storage system's fault-tolerance and reliability by reducing disk reconstruction time when a hard disk or a flash disk fails. For example, when a hard disk fails, its partner flash disk can largely help the recovery of the failed hard disk in two ways. First of all, since part of data was on the flash disk, the replacement hard disk only needs to recover the data that was originally on the failed hard disk. Second, during the hard disk reconstruction process, the flash disk can still serve part of normal requests from outside clients, which greatly alleviates the workload of the replacement hard disk, which in turn speeds up the disk recovery process. Thus, the hard disk reconstruction time in the proposed hybrid disk storage architecture is shorter than that of in a pure hard disk array architecture. The SAIL strategy is implemented as a software component within the hybrid disk array controller. It consists of the following five modules: file placement initializer, file access monitor, redistribution table generator, file re-organizer, and disk space manager. The five modules coordinate together to dynamically allocate and reallocate files between the hard disk array and the flash disk array (Fig. 2).

3 The SAIL Strategy

The methodology behind the SAIL strategy is to judiciously yet adaptively divide the entire file set into a *flash-preferred* subset and a *hard-preferred* subset based on dynamic I/O workload characteristics. Each subset of files is then allocated onto its favorite disk array so that the complementary merits of flash disks and hard disks can be mostly utilized while their respective disadvantages can be largely avoided. Supported by the proposed hybrid disk storage architecture, the goal of SAIL is to achieve a high performance, energy conservation, and desirable system reliability at the same time.

3.1 Design Methodology

SAIL realizes its goal by exploiting two critical I/O workload characteristics: file access locality and file access type. The presence of access locality in I/O workload has long been recognized in the literature. For example, it is well-known that 10% of files accessed on a web server account for 90% of the server requests and 90% of the bytes transferred [1]. Similar workload locality has also been observed in OLTP applications running in large financial institutions [11]. The implication of workload locality is that the overall system performance can be noticeably improved if the I/O requests on the small percentage popular files can be served more efficiently. File access locality suggests us concentrate on the allocation and reallocation of the minority popular files. The second important I/O workload characteristic is file access type, namely, write-excessive, read-exclusive, and read-write. In an investigation of file system workloads, Roselli et al. found that file access has a bimodal distribution pattern within which some files are written excessively without being read while others are almost exclusively read [19]. This observation confirms that it is feasible for SAIL to separate files into the aforementioned three categories based on the type of accesses that they received. It is easily understood that read-exclusive files are suitable for flash disks as they don't contribute any erasure cycles to flash disks. Further, accessing these read-exclusive files on flash disk can significantly save energy and gain potential performance enhancement due to no seek time and rotation latency any more. Similarly, write-excessive files are more appropriate for hard disks where erasure cycle limitation doesn't apply. The most difficult task for a file allocation and reallocation strategy is to decide where some read-write popular files should go. Unlike existing conservative algorithms such as PB-PDC [15][17], which immediately puts all read-write files onto hard disks to avoid any write cycles on flash disk, SAIL adopts a more open attitude and makes a smart decision based on a good trade-off between performance and energy saving.

3.2 System Models

The set of files is represented as $F = \{f_1, ..., f_i, \dots, f_m\}$. Also, the flash disk array is modeled as $FD =\{fd_1, ..., fd_j, \dots, fd_n\}$, whereas the hard disk array is denoted by *HD*

$=\{hd_1, ..., hd_j, \ldots, hd_n\}$. Since each file will be allocated onto either a set of hard disks or a set of flash disks in a striping manner, let sp denote the size of a stripe in Kbyte and it is assumed to be a constant in the system. A file f_i ($f_i \in F$) is modeled as a set of rational parameters, e.g., $f_i = (s_i, r_i, w_i, b_i)$, where s_i is the file's size in Mbyte, r_i is the file's read access rate (1/second), w_i is the file's write access rate (1/second), and b_i is the number of batches of the file, which is defined in Eq. 1. Assume that d_i^{start} is the starting disk of file f_i's striping distribution.

$$b_i = \begin{cases} 1, \text{if } s_i/sp \le (n - d_i^{start} + 1) \\ 1 + \left\lceil ((s_i/sp) - (n - d_i^{start} + 1))/n \right\rceil, \text{ otherwise} \end{cases} . \tag{1}$$

Each hard disk's transfer rate (for both read and write) is t^h (Mbyte/second). For both hard disks and flash disks, we only consider a two- level power model: active mode and idle mode. In other words, a flash disk or a hard disk can only work in either active mode when it reads/writes data or idle mode when no such read/write activities occur. This assumption is valid for server-class applications because intensive server-level workload does not allow hard disks to spin up/down to save energy due to very slim time slots between requests. A hard disk's active energy consumption rate and idle energy consumption rate are p^h (Watts) and i^h (Watts), respectively. Similarly, a flash disk is modeled as $fd_j = (r^f, w^f, p^f, i^f)$, where r^f is its read rate (Mbyte/second), w^f is its write rate (Mbyte/second), p^f is its active energy consumption rate (Watts), and i^f is its idle energy consumption rate (Watts). In addition, *SK* denotes average seeking time of a hard disk and *RT* represents average rotation latency of a hard disk. The time span of one epoch is denoted by T_e (second). Therefore, the mean service time of file f_i served by a hard disk is

$$mst_i^h = b_i (SK + RT + sp / t^h) . \tag{2}$$

However, if the file is served by a flash disk, its mean service time becomes

$$mst_i^f = [r_i T_e b_i (sp / r^f) + w_i T_e b_i (sp / w^f)]/(r_i + w_i) T_e = bi(r_i / r^f + w_i / w^f)/(r_i + w_i) . \tag{3}$$

Hence, the performance gain pg_i in terms of mean service time reduction ratio of file f_i is defined in Eq. 4.

$$\begin{aligned} pg_i &= mst_i^h / mst_i^f \\ &= (SK + RT + sp / t^h)(r_i + w_i)/(r_i / r^f + w_i / w^f) \end{aligned} . \tag{4}$$

For each read-write file, we need to decide where to store it. Thus, we need to calculate its energy gain eg_i in one epoch in Eq. 5, where ec_i^h is the energy consumption of file f_i in one epoch if it is stored in the hard disk array, and ec_i^f is the energy consumption of f_i in one epoch if it is in the flash disk array.

$$eg_i = ec_i^h / ec_i^f = [\frac{p^h}{t^h}(r_i + w_i)]/[p^f (\frac{r_i}{r^f} + \frac{w_i}{w^f})] . \tag{5}$$

Since in some situations it is desirable to trade performance for energy-saving, SAIL employs a parameter named PDA (*performance degradation allowed*) to make a good trade-off between performance and energy when it makes reallocation decisions for read-write files. Essentially, PDA is a constant value set by system administrator and it is in the range [0, 1). If the system administrator believes that performance is the most important goal, he can set PDA as zero, which implies that performance degradation caused by allocating a read-write file onto the flash disk array is not permitted. If sacrificing some performance for energy-saving is desirable, he can set PDA to a value larger than zero (e.g., 20%). In this case, if a read-write file's performance gain pg_i (Eq. 4) is within the range [1-PDA, 1] and its energy gain eg_i (Eq. 5) is larger than 1, the file will be reallocated onto the flash disk so that energy-saving can be realized at the price of performance.

The total number of write cycles of a flash disk is a constant *WC*, which is assumed to be 1 million in our simulation experiments. Besides, *DY* represents the duration years of a flash disk and we set *DY* as 5 years. As a result, *WCPS* (write cycles per second) that is allowed by a flash disk is defined in Eq. 6 as below.

$$WCPS = (WC / DY)/(365 * 24 * 60 * 60) . \tag{6}$$

For instance, the value of *WCPS* in our simulations is around 0.0063 (1/second). Therefore, the reliability loss rl_i of file f_i if it is stored on the flash disk array can be computed by

$$rl_i = \begin{cases} 1, \text{ if } w_i \geq WCPS \\ 0, \text{ otherwise} \end{cases} . \tag{7}$$

The request set is designated as $R = \{r_1, ..., r_k, ..., r_x\}$. Each request is modeled as $r_k = (fid_k, a_k, t_k)$, where fid_k is the file ID that is accessed by the request r_k, a_k is the arrival time of request r_k, t_k is the type of the request r_k and it can be "*r*", "*w*", "*c*", and "*d*" representing "read", "write", "create", and "delete", respectively.

3.3 Implementations

The SAIL strategy consists of five modules (Fig. 2) that coordinate with each other via five data structures (Fig. 3): file position and popularity table, file re-distribution table, free flash space queue, free hard space queue, and deleted file queue.

At the beginning, all files are striped across the hard disk array in a RAID-0 fashion. Dynamically created files are also distributed initially across the hard disk array. SAIL first starts the file placement initializer, which creates some important data structures such as file position and popularity table for later use (Fig. 3). After the hybrid disk array begins to serve I/O requests, SAIL launches the file access monitor to record each file's popularity in terms of number of accesses within one epoch in the file position and popularity table. The file position and popularity table, which contains the latest popularity information of each file, will be used later by the redistribution table generator to generate the file re-distribution table. After labeling all popular files, the redistribution table generator generates the file re-distribution table, which lists all files that need to be reallocated between the hard disk array and the flash disk array. Guided by the file re-distribution table, the file re-organizer reallocates all files in the file re-distribution table to their preferred destinations. During the file reallocation process, the file re-organizer consults to the disk space manager, which is responsible for managing disk space for both hard disk array and flash disk array.

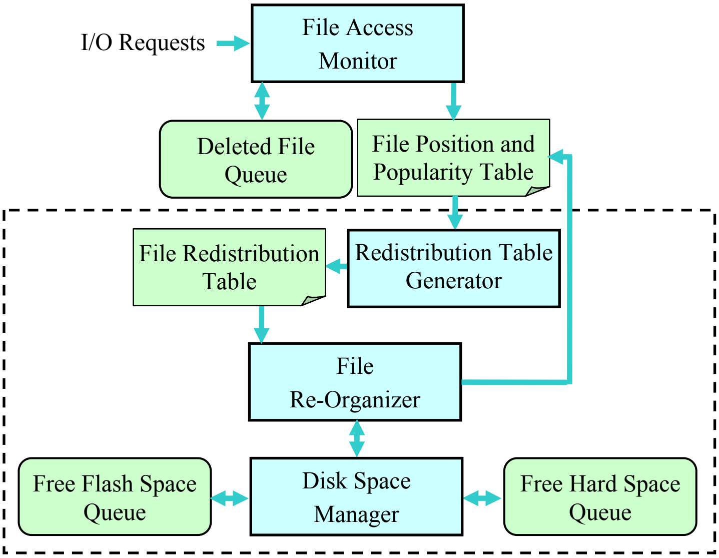
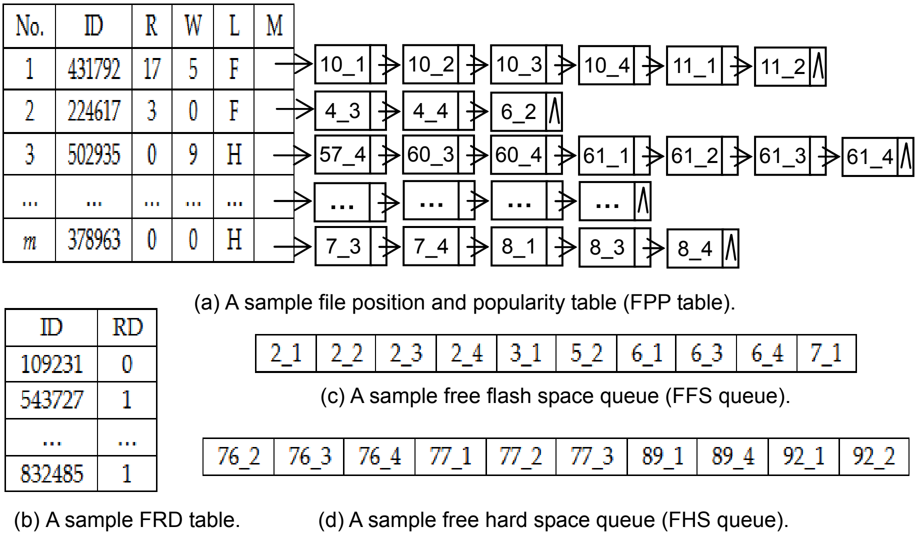


Fig. 2. The SAIL strategy; modules in the dotted rectangle execute once per epoch



No.	ID	R	W	L	M
1	431792	17	5	F	
2	224617	3	0	F	
3	502935	0	9	H	
...	...	...	...	...	
m	378963	0	0	H	

(a) A sample file position and popularity table (FPP table).

ID	RD
109231	0
543727	1
...	...
832485	1

(b) A sample FRD table.

2_1	2_2	2_3	2_4	3_1	5_2	6_1	6_3	6_4	7_1

(c) A sample free flash space queue (FFS queue).

76_2	76_3	76_4	77_1	77_2	77_3	89_1	89_4	92_1	92_2

(d) A sample free hard space queue (FHS queue).

Fig. 3. Major data structures for the SAIL strategy

Obviously, file reorganization is achieved at the cost of both performance degradation and extra energy consumption. Fortunately, SAIL only needs to re-organize a small portion of popular files at the end of each epoch due to the smooth changes in file access pattern. Also, to reduce the overhead associated with file re-organization, SAIL confines the time span of each epoch so that frequent file reallocation can be avoided.

4 Performance Evaluation

This section presents results of a comprehensive experimental study comparing the proposed SAIL strategy with the PB-PDC algorithm. To the best of our knowledge, PB-PDC is the only existing data placement algorithm that partitions data between a hard disk and a flash memory device. This is largely because how to combine newly manufactured flash disks with traditional hard disk drives to form efficient storage systems for data-intensive applications is a brand new research topic. Note that one of the most significant differences between SAIL and PB-PDC is that SAIL is integrated with RAID structures on top of a hybrid disk array for enterprise applications, whereas PB-PDC in its current status merely employs one hard disk with one flash memory device in a personal laptop computing environment. In this section, we first introduce experimental setup including performance metrics, the real trace, hard disk and flash disk characteristics, and simulation parameters that we used. Next, in Sections 4.2 we analyze experimental results.

4.1 Experimental Setup

We developed an execution-driven simulator that models a hybrid disk array, which has one hard disk array and one flash disk array (see Fig. 1). The main characteristics of the hard disk and the flash disk used in simulations are shown in Table 1. The performance metrics by which we evaluate system performance include:

- *Mean response time*: average response time of all access requests submitted to the simulated hybrid disk array storage system.
- *Energy consumption*: energy consumed by the hybrid disk array during the process of serving the entire request set.
- *Write cycles*: the maximal number of write on one flash disk during one epoch.

We evaluate the SAIL and the PB-PDC algorithms by running trace-driven simulations over the Auspex trace originated from Berkeley [8], which has been widely used in the literature. Since the simulation times in our experiments are much shorter compared with the time span of the trace, we only choose the first 1100,000 I/O requests from the trace in our experiments. We examined the impacts of flash disk capacity on system performance by controlling the parameter.

Table 1. Hard disk and flash disk parameters

Hard disk	Seagate Cheetah 15K.4	**Flash disk**	Adtron Flashpak
Model number	ST373454FC	Model number	A25FB-20
Capacity (GB)	73.4	Capacity (GB)	4, 8, 16, 24, 32
Spindle speed (RPM)	15 K	Access time (ms)	0.272
Ave. seek time (ms)	3.5	Seek time	0
Ave. latency (ms)	2.0	Read (Mbytes/sec)	78
Transfer rate (Mbytes/sec)	77	Write (Mbytes/sec)	47
Active power (watts)	17	Read/write power (watts)	3.43
Idle power (watts)	11.4	Idle power (watts)	1.91

4.2 Impact of Flash Disk Capacity

The first group of experiments was conducted to study the impact of flash disk capacity on the performance of the two algorithms (Fig. 4). An average improvement of 24.2% in mean response time and 28.2% in energy consumption were observed by SAIL over PB-PDC (Fig. 4).

With an increased capacity of each flash disk, it is easy to understand that both SAIL and PB-PDC can improve their performance in terms of mean response time (Fig. 4). Meanwhile, energy consumption of both algorithms is reduced (Fig. 4). This is because more popular files can be placed on flash disks when more flash disk space is available. In terms of the maximal write cycles on one flash disk during an epoch, SAIL results in only 35 write cycles within one epoch (1000 seconds) when the capacity of a flash disk is 16 GB. Considering the huge capacity of a flash disk and the relatively very small number of total write cycles on it, the write cycles per block within one epoch caused by SAIL is far from a flash disk's write cycle threshold value *WCPS* (see Eq. 6). Besides, modern flash disks normally have built-in wear-leveling techniques [21]. Thus, we believe that the impact of SAIL on flash disk reliability can be safely omitted.

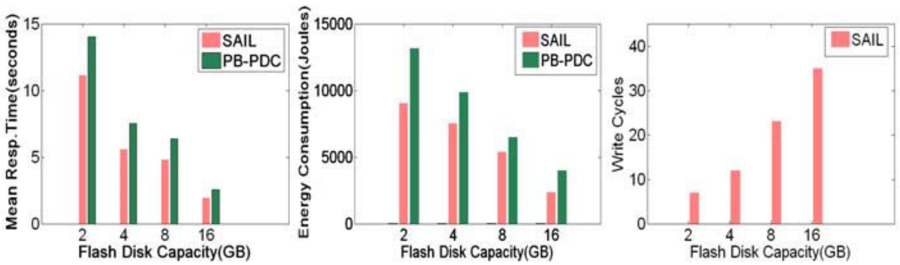


Fig. 4. An overall comparison of the two algorithms with respect to flash disk capacity

5 Conclusions

In this paper, we address dynamic file allocation and reallocation problem in the context of a hybrid disk array. A new disk array architecture was proposed to replace traditional pure hard disk based disk arrays in server-class data-intensive applications. Powered by the proposed storage architecture, we further designed and implemented a novel self-adaptive dynamic file allocation and reallocation strategy SAIL, which judiciously separate files between one hard disk array and one flash disk array based their access patterns. Thus, the complementary merits of hard disk and flash disk can be mostly utilized while their respective shortcomings can be avoided. Comprehensive simulation experiments demonstrate that SAIL consistently outperforms an existing dynamic file assignment algorithm PB-PDC, which also employs both hard disk and flash device. Specifically, our trace-driven experimental results show that the SAIL strategy results in an average 24.2% and 28.2% performance and energy consumption improvement compared with PB-PDC. Meanwhile, in terms of write cycles, SAIL guarantees that its impact on flash disk reliability is trivial and can be safely ignored.

Acknowledgments. This work is partially supported by the US National Science Foundation under grants CNS-0834466 and CCF-0742187.

References

1. Arlitt, M., Williamson, C.: Web server workload characterization: the search for invariants. In: ACM SIGMETRICS Conference, pp. 126–137. ACM Press, New York (1996)
2. Arnan, R., Bachmat, E., Lam, T.K., Michel, R.: Dynamic data reallocation in disk arrays. ACM Transactions on Storage 3(1), 2 (2007)
3. Bisson, T., Brandt, S.: Reducing energy consumption with a non-volatile storage cache. In: International Workshop on Software Support for Portable Storage, New York (2005)
4. Cash, K.: Flash Solid State Disks - Inferior Technology or Closet Superstar? BiTMICRO Networks, `http://www.storagesearch.com/bitmicro-art1.html`
5. Chang, L.P., Kuo, T.W.: Efficient management for large-scale flash-memory storage systems with resource conservation. ACM Transactions on Storage 1(4), 381–418 (2005)
6. Cheetah 15K.4 Mainstream enterprise disc drive storage, `http://www.seagate.com/content/docs/pdf/marketing/Seagate_Cheetah_15K-4.pdf`
7. Chen, F., Jiang, S., Zhang, X.: SmartSaver: Turning Flash Drive into a Disk Energy Saver for Mobile Computers. In: International Symposium on Low Power Electronics and Design, pp. 412–417. IEEE Press, New York (2006)
8. Dahlin, M.D., Wang, R.Y., Anderson, T.E., Patterson, D.A.: Cooperative caching: using remote client memory to improve file system performance. In: USENIX Operating Systems Design and Implementation, vol. 1, Article No. 19 (1994)
9. Dowdy, W., Foster, D.: Comparative Models of the File Assignment Problem. ACM Computing Surveys 14(2), 287–313 (1982)
10. Fitzgerald, A.: Flash Disk Reliability Begins at the IC Level. COTS Journal, `http://www.cotsjournalonline.com/home/article.php?id=100053`
11. Goyal, P., Jadav, D., Modha, D.S., Tewari, R.: CacheCOW: providing QoS for storage system caches. In: SIGMETRICS Conference, pp. 306–307. ACM Press, New York (2003)
12. Hsieh, J.W., Kuo, T.W., Chang, L.P.: Efficient Identification of Hot Data for Flash Memory Storage Systems. ACM Transactions on Storage 2(1), 22–40 (2006)
13. Kawaguchi, A., Nishioka, S., Andmotoda, H.: A flash-memory-based file system. In: USENIX Technical Conference, pp. 155–164 (1995)
14. Kim, H., Lee, S.G.: A new flash-memory management for flash storage system. In: The 23rd International Computer Software and Applications Conference, pp. 284–289 (1999)
15. Kim, Y.J., Kwon, K.T., Kim, J.: Energy-efficient file placement techniques for heterogeneous mobile storage systems. In: The 6th ACM & IEEE International Conference on Embedded Software, pp. 171–177 (2006)
16. Lee, L.W., Scheuermann, P., Vingralek, R.: File assignment in parallel I/O systems with Minimal Variance of Service Time. IEEE Transactions on Computers 49(2), 127–140 (2000)
17. Pinheiro, E., Bianchini, R.: Energy Conservation Techniques for Disk Array-Based Servers. In: International Conference for High Performance computing, Networking, Storage and Analysis (Supercomputing 2004), pp. 88–95 (2004)
18. Product Specification, Adtron A25FB-20 Flashpak Data Storage, `http:// www.adtron.com/pdf/A25FB-20-sum052908.pdf`

19. Roselli, D., Lorch, J.R., Anderson, T.E.: A Comparison of File System Workloads. In: USENIX Technical Conference, pp. 44–54 (2000)
20. Scheuermann, P., Weikum, G., Zabback, P.: Data partitioning and load balancing in parallel disk systems. The International Journal on Very Large Data Bases 7(1), 48–66 (1998)
21. Storage Products, A25FB-20-R2spec101507.pdf
22. Weikum, G., Zabback, P., Scheuermann, P.: Dynamic file allocation in disk arrays. ACM SIGMOD 20(2), 406–415 (1991)
23. Xie, T.: SOR: A Static File Assignment Strategy Immune to Workload Characteristic Assumptions in Parallel I/O Systems. In: The 36th International Conference on Parallel Processing. IEEE Press, New York (2007)
24. Xie, T., Sun, Y.: No More Energy-Performance Trade-Off: A New Data Placement Strategy for RAID-Structured Storage Systems. In: The 14th Annual IEEE International Conference on High Performance Computing, pp. 35–46. Springer, Heidelberg (2007)

Directory-Based Conflict Detection in Hardware Transactional Memory

Rubén Titos, Manuel E. Acacio, and José M. García

Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia, 30100 Murcia, Spain
{rtitos,meacacio,jmgarcia}@ditec.um.es

Abstract. One of the key design points of any hardware transactional memory (HTM) system is the conflict detection mechanism, and its efficient implementation becomes critical when conflicts are not a rare event. While many contemporary proposals rely on the coherence protocol to carry out conflict detection at the private cache levels, this approach is not optimal for systems that use a directory to maintain coherence over an unordered, scalable network, such as tiled CMPs. In this paper, we present a new scheme of conflict detection for HTM systems, which moves this key mechanism from the private caches to the directory level. We propose a novel transactional book-keeping method and describe how this detection can be carried out more efficiently at the directory. Simulation results show that our approach obtains reductions in execution time between 25 and 55% for transactional benchmarks with a high number of conflicts, with an average improvement over LogTM-SE of 15%.

1 Introduction

Transactional Memory (TM) has arisen as a promising programming model targeted to ease parallel programming while still producing efficient multithreaded programs that exploit the computational resources available in present and future multicore chips. Using the TM model, the programmer declares *what* regions of the code must appear to execute in mutual exclusion, leaving the burden of *how* to provide atomicity and isolation to the underlying levels. The system then optimistically executes transactions, stalling or aborting them whenever real runtime data conflicts appear. Although a TM system can be entirely implemented in software, moving some basic transactional functionality to the hardware level is essential to minimize its performance overhead. This paper focuses on hardware TM systems (HTMs) whose aim is to bring the TM model to the high-performance computing arena.

One of the key mechanisms of any transactional system is conflict detection. A conflict occurs when two or more concurrent transactions access the same data block and at least one of the accesses is a write. In order to detect such violations of isolation, a TM system must keep track of its transactions' read and write sets. Some proposed HTMs perform this book-keeping by extending each cache

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 541–554, 2008.

entry with R/W bits [4][8]. Other designs opt for per-thread hash signatures to encode address sets using Bloom filters [12].

Regardless of how the TM system records R&W sets, another major design dimension of conflict detection is *when* to use this information to check for violations of isolation. This can be done either immediately after every memory request – eager policy – or it can be delayed until the end of the transaction – lazy detection –. Most HTM systems proposed to date implement eager conflict detection by modifying standard ownership-based cache-coherence protocols [5] [1][9][8][12]. These systems monitor the coherence traffic for transactional blocks to determine if another processor is performing a conflicting access.

Semantically, a transaction must retain exclusive ownership over its written blocks, and non-exclusive ownership over its read blocks, until it reaches commit. Because ownership is usually associated with cache residence, any coherence protocol capable of detecting ownership conflicts can also detect transaction conflicts at no extra cost. However, limited cache capacity and associativity lead to replacements of active transactional blocks, thus breaking the ownership-cache residence connection that basic conflict detection relies upon. In order to support transactions of an arbitrary size, HTMs should ensure isolation in the presence of overflowed transactional blocks (evicted from the private cache level). Thus, a transactional node needs to see the coherence traffic for blocks that are no longer locally cached. While this happens naturally in systems with snoopy-based cache coherence, like the original TM proposal by Herlihy and Moss [5], it constitutes an abnormal behaviour for a directory-based protocol that maintains coherency over an unordered, point-to-point network, as that of a tiled CMP.

The introduction of the so-called *sticky states* in directory-based protocols [8][12] basically consists of using the directory entry to track the current transactional owner of an evicted block, and forward requests for that block to the transactional owner so that it can detect conflicting accesses that try to revoke its transactional ownership. Somehow, this can be regarded as a timid first step towards the fusion of cache coherence and conflict detection. Such combination of two seemingly independent mechanisms is not new, but it was already a fundamental part of TCC [4], an HTM in which lazy conflict detection and snoopy coherence were merged to provide a consistency model based on transactions.

In this paper, we propose a novel approach to eager conflict detection that further extends a directory protocol in order to provide a fast detection scheme in tiled CMP architectures. By comparing our proposal with an HTM system such as LogTM-SE [12], we observe several advantages of implementing conflict detection at the directory level instead of at the cache level. First and foremost, detection itself is accelerated, as conflicts are always detected in one hop instead of two. Considering that one of TM's fundamental principles is to achieve programming ease by allowing coarse-grained transactions, it is of great importance that conflicts are handled as efficiently as possible, as they are likely to occur often when the programmer relies on large transactions. Indeed, most of the transactional workloads from the Stanford Transactional suite (STAMP) [3] already pose this high-conflict behaviour, as shown in [10]. Second, by detecting conflicts faster, the proposed TM system reacts more rapidly to high-contention scenarios and has the potential to avoid many aborted transactions, improving performance. Simulation results using GEMS (*General Execution-driven Multiprocessor Simulator*) show that our conflict detection approach obtains reductions in execution time of 15% on average for the selected benchmarks, with better performace gains – up to 55% – for those workloads that suffer frequent transaction conflicts.

The rest of the paper is organized as follows: Section 2 briefly describes the different approaches to conflict detection adopted by some of the most relevant contributions to hardware transactional memory, and motivates our work. In Section 3 we describe our directory-based conflict detection scheme. Section 4 evaluates the performance of our proposal, comparing it to an ideal LogTM-SE system. We end with Section 5, which summarizes the main conclusions of this study and presents our future work.

2 Motivation and Related Work

In the early nineties, Herlihy and Moss introduce *Transactional Memory* (TM) [5] as a hardware alternative to lock-based synchronization. Their proposal relies on a snoopy coherence protocol to detect conflicting accesses, providing atomic accesses to several independent memory locations. More than a decade later, Hammond *et al.* present TCC, *Transactional Coherence and Consistency* [4], a novel coherence and consistency model based on transactions. The TCC system is also built upon a broadcast network that allows transactions to snoop commit traffic to maintain coherence and detect possible dependence violations (conflicts). Later on, several proposals such as UTM [1] or VTM [9] focus on hardware schemes that provide virtualization of transactions, i.e., support for transactions of unlimited duration, size and nesting depth. Both UTM and VTM monitor the coherence traffic for the transaction's cache lines to determine if another processor is performing a conflicting operation. In LogTM [8], Moore *et al.* combine transactional support with a conventional shared memory model, also taking the coherence protocol as a means to perform conflict detection. LogTM-SE [12] is a subsequent refinement that decouples transactional support from caches using hash signatures to detect conflicting threads.

Some of these HTM proposals perform transactional book-keeping by extending each cache entry with R/W bits [4][8]. Despite losing the information needed to perform conflict detection when when transactional blocks are evicted from the cache, these systems manage to guarantee isolation in this circumstances at a performance cost. On one hand, TCC [4] enforces transaction serialization by letting a transaction write its results directly to shared memory. On the other hand, LogTM [8] lets blocks leave the cache, and modifies a directory coherence protocol with *sticky states* so that the overflowed cache keeps receiving forwarded requests and performing conflict detection on the evicted blocks. LogTM's approach of lazily cleaning up sticky states suffers from frequent false positives when overflows become more frequent, due to stale directory information. Other

HTM designs opt for per-thread hash signatures to encode address sets using Bloom filters [12][3]. Under this alternative, transactional blocks that overflow cache are no longer a problem, as the information needed to detect conflicts is decoupled from the data block and stored at the core level. However, due to their conservative encoding, hash-signatures may signal a conflict when none exists (a false positive), causing unnecessary rollbacks that degrade performance. The ratio of false positives becomes significant when the transaction footprint grows, disencouraging the programmer from using coarse grain synchronization and somehow jeopardizing one of the main goals of TM.

2.1 Why Detect at the Directory Level

Up until now, conflict detection has always been performed at the private cache levels of the memory hierarchy. This makes the most sense when private caches are able to snoop on every memory transaction that takes place across the system, by being connected to a shared, ordered network like a bus [5][4]. However, in more scalable networks where directory-based protocols are more appropriate to maintain coherence, a cache only observes the requests for those blocks that are locally cached. Despite this substantially different scenario, eager conflict detection schemes that rely on directory protocols have so far implicitly inherited the same style of *private-level* conflict detection [8][12].

In this context, a reason why the directory is best suited for conflict detection is its location. From the perspective of a memory transaction, L1 caches are *end-points* – a request's origin or destination – whereas the L2 directory acts as a *middle-point* that orchestrates the traffic – routing requests so that they arrive at their destination –. As end-points entities, L1 caches are not a straightforward location to perform conflict detection: For them to detect conflicts on their evicted transactional blocks, the directory needs to behave abnormally and forward requests for blocks that are no longer cached at the private level. The directory, however, is not only a middle point that naturally observes all the traffic for its mapped blocks, but also the *first stop* of any request message, thus becoming the perfect location to provide a fast (one-hop) detection scheme.

Besides its privileged location, the directory's role makes conflict detection a simple addition to its responsibilities. Considering that i) the directory is in charge of tracking each cached block's ownership[1], and that ii) transactional ownership is connected to cache residence in the common case (except for evicted transactional blocks), the directory has most of the information required to detect memory accesses that attempt to revoke a node's transactional ownership over its read and written blocks. Therefore, such an extension in its functionality becomes a natural evolution of its role within a TM system.

Furthermore, an HTM with directory-based conflict detection also mitigates the performance implications of signature false positives. In systems where not every memory block has its corresponding directory entry, the directory controller still needs to keep detecting conflicts for those transactional blocks that

[1] We use the term ownership throughout this paper to stand for cache residence, independent of the state of the block in the cache(s).

are spilled from the coherence level. Per-bank signatures in the directory are a good solution to this problem because the number of transactional active blocks that overflow this level (i.e. the L2 cache in a CMP) is insignificant in comparison to total number of blocks accessed by a transaction, and so is the probability of false positives, compared to using signatures to encode the entire access set.

3 Directory-Based Conflict Detection

Using the directory to check for conflicts over blocks that remain cached by transactional owners does not need any more information about a block than what is already stored in its directory entry. For example, let W be a transactional writer that locally caches a block B with exclusive ownership, and let R be a reader that tries to acquire non-exclusive ownership of B. When R's read request arrives to the directory, the standard protocol dictates that the request must be forwarded to W, which would then detect the conflict. However, if the directory only *knew* that W is executing a transaction, forwarding the request to W would be unnecessary; the directory itself could immediately detect a conflict on B and take the appropriate actions to resolve it. To do this, the directory only needs to keep a record of which cores are executing a transaction at any moment. To this end, our base conflict detection scheme explicitly notifies the directory about transaction begin and transaction commit. A simplistic solution could consist of sending dedicated begin/commit messages and waiting for acknowledgment before resuming the execution.

Once the directory knows that a core P is executing a transaction, it could immediately start to detect conflicts for all accesses to blocks locally cached by P. However, doing so would lead to many unnecessary conflicts since not all cached blocks may have been accessed by the transaction – in other words, cache residence does not necessarily imply transactional ownership –. In order to avoid them, the naive approach of our base scheme is flush-clearing the local data cache at transaction begin, writing back all modified/exclusive blocks. Following this simple approach, the first reference to each data block from inside a transaction misses in the local cache, so that the directory observes all transactional addresses and perform the book-keeping required for conflict detection. While flush-clearing the data cache is clearly not desirable, those workloads composed of large transactions should not be too affected, as flushes happen infrequently. The main drawback of flushing appears in applications with short, frequent transactions, in which not only the transaction but also the following code, find *an almost* empty data cache. Most misses suffered by the post-transactional code (which presumably operates on local data) are directly caused by the recent cache-flush. For this reason, more sophisticated schemes are necessary, which would allow for conflict detection at the directory level without requiring a cache-flush on every transaction begin.

So far, we have assumed in our elaboration that transactional ownership implies cache residence, but that is not always the case because transactional blocks can exceed the capacity or associativity of the local cache. Since no explicit information about a transaction's R&W sets is stored at the core level (no signatures), the directory needs to track transactional blocks that are evicted from the private level while the transaction runs, in order to keep detecting conflicting accesses on those blocks. To this end, we introduce the concept of *Transaction Serial Number* (XSN), a small, reusable, *per-core* identifier that is used by the directory to tag transactional blocks and maintain a correspondence between a block and its owner transaction(s). While we have not determined the ideal size of the transaction serial number, performing lazy clean-up of non-matching XSNs greatly reduces the overhead of these identifiers. A few bits per XSN should suffice to avoid virtually all false conflicts due to XSN reuse. Nonetheless, these false positives caused by stale XSNs that become *fresh* only affect the performance but not the correctness of the transactional execution.

Hardware Requirements. On the core side, each core has a counter (*XSN register*) that contains the XSN assigned to its last/current transaction. The *XSN register* is incremented every time the instruction begin_transaction is executed – hence also after an abort –. Its content is copied to all the outgoing transactional messages (set to zero for all non-transactional requests), allowing the directory to differentiate between transactional and non-transactional requests. On the directory side, each directory bank keeps a vector of XSN's (*global XSNs*), one XSN per core. The corresponding XSN of the vector is updated on every begin_transaction with the XSN indicated in the message, while it is set to zero ("not in transaction") upon arrival of a commit_transaction message. As for the directory entry, each one is augmented with a new field, *xact owners XSN*, whose function is to keep a correspondence between the block and its current transactional owner(s). For simplicity, we can think of this field as a vector with as many XSN as cores. In practice, each entry does not need to store one XSN per core; instead, the hardware overhead of this mechanism can be minimized by having a separated XSN buffer that the directory controller uses "on demand". Finally, the directory uses a set of per-core signatures to track those transactional active blocks that are evicted from the directory level. Before the replacement, the address is added to the signature of its transactional readers/writer. These signatures are only checked in case a request misses at the coherence level, and cleared on transaction commit/abort.

Operation. By jointly considering both a block's XSNs and the global XSN vector, the directory can unequivocally determine if a certain block is owned by some currently running transaction(s) or if, on the contrary, some transaction that made use of it has committed/aborted. The basic idea behind this mechanism is that a block is considered part of a transaction's R/W set running in P when the P-th XSN of its *xact owners* matches the P-th XSN of the global XSN vector. Comparing a block's XSN against the global XSN vector, the directory tracks transactional ownership even when the block is not privately cached, enabling conflict detection regardless of the actual location of the block. Figure 1 illustrates the proposed conflict detection mechanism, showing how the forementioned hardware elements work together to provide fast conflict detection at the

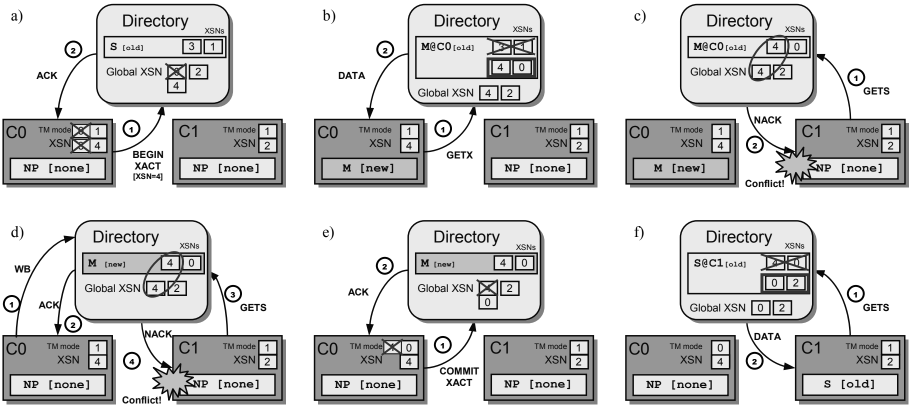


Fig. 1. Examples of Directory-Based Conflict Detection

directory level. The figure also shows the coherence state for one block at the directory and in two core's private caches, as well as the block's XSNs. The directory's global XSN vector and each core's XSN register are also shown.

Core 0 (C_0) begins its transaction by incrementing its XSN register and sending it to the directory through an explicit *begin_transaction* message (Figure 1 a). The directory uses this message to update its global XSN register and responds with an acknowledgment, allowing the core to begin its transaction. In Figure 1 b, C_0 attempts to write a block, but misses in its private cache and sends an exclusive request to the directory. The directory checks the block's *xact owners XSN*, observes that the reader transaction in C_1 is no longer running and sends exclusive data to C_0, setting both the state and *xact owners* accordingly – lazily clearing C_1's stale XSN –. In Figure 1 c, the transaction in C_1 tries to read the same block, missing in its L1. Comparing *xact owners* and the global XSN, the directory finds out that C_0 is a transactional owner, and then it uses the coherence state to find out whether C_0 is a reader or a writer. In this case, the block is not in shared state, which means C_0 is a writer and hence the directory detects the conflict. Figure 1 d is an example of the out-of-cache conflict detection: C_0 writebacks the block and when C_1 retries its read request, the directory detects the conflict once again, since the writeback did not change the xact owners. At last, C_0 commits its transaction, notifies the directory (Figure 1 e), allowing C_1 to finally obtain a shared copy of the block (Figure 1 f).

3.1 Enhancements to the Base Detection Scheme

Augmenting the Private Cache to Avoid Flushing. Instead of flush-clearing the L1 cache on every transaction begin, a more elaborated solution could serve those accesses that hit on privately cached data immediately, and allow the core to continue its execution without any extra delay, while sending a notification down to the directory (off the critical path). This *report* messages contain the new transaction serial number of the just-started transaction and are used to update the block's *xact owners XSN* vector at directory. A *Transactional* bit must be added to each L1 cache line, to deal with forwarded conflicting request as well as to reduce the number of reports sent down to the directory. This bit is set each time a block is accessed and flush-cleared on transaction commit. Report messages are only sent out if the bit is not set. If a race occurs between a remote request and a report message, so that the remote message arrives before at the directory, the core receives the forwarded request and it signals a conflict if it finds the *Transactional* bit set for the block. Eventually, the directory information for that block will be updated with the new XSN and subsequent conflicting requests will be handled entirely at the directory level.

Reporting Begin/Commit to the Directory without Extra Delay. Instead of sending one begin and one commit message to each directory bank for each transaction, a more scalable solution could use *on-demand piggybacking* for these reports. This can be done by recording which L2 banks the core has accessed during the transaction, using a simple bit-vector that is updated by the address-to-bank mapping logic on each L1 miss and cleared after commit. In this way, the begin transaction report is inserted as a field (XSN) in the first request message sent to a directory bank, without delaying the execution of the transaction. At transaction commit, only the appropriate directory banks need to be notified, according to the forementioned bit-vector.

4 Evaluation

In this section, we evaluate the performance of the proposed conflict detection scheme (DirCD). We use the LogTM-SE hardware transactional memory system as the basis of our simulations, and we modify it to introduce two versions of our proposal: a naive implementation that empties the L1 cache on every begin_transaction (*DirCD+L1Flush*), and an enhanced version that avoids cache flushing (*DirCD+NoL1Flush*). To provide a better perspective over the results, we also consider an identical configuration to the baseline LogTM-SE system that flush-clears the L1 cache on transaction begin (*CacheFlush*). We compare these two DirCD flavours against an ideal configuration of LogTM-SE in which perfect signatures are used to track R/W sets and detect conflicts (*Base*).

For simplicity, our version of *DirCD+NoL1Flush* does not use *hit-report* messages nor L2-overflow signatures; instead, we approximate a *flush-free* configuration by relying on the original address signatures of LogTM-SE, which we have made directly accessible to the directory conflict detection logic. The resulting implementation emulates a more sophisticated DirCD-based system, which incorporates the enhancements described in 3.1. Regarding conflict resolution (CR), it is now performed at the directory level, although the CR policy remains fixed – requester stalls, with conservative deadlock avoidance –. Our DirCD implementation also reuses the functionality that the simulator provides for LogTM-SE, so that the directory does not track timestamps, possible cycles nor does it issue abort messages when a possible deadlock is detected. Lastly, our conflict detection scheme is evaluated without restricting the size of transaction serial numbers, and using a full XSN-vector in each directory entry.

4.1 Summary of LogTM-SE

LogTM is a hardware transactional memory system proposed by the Multifacet group at the University of Wisconsin-Madison. LogTM implements eager version management and eager conflict detection. It uses a per-thread log in cacheable virtual memory that contains address and old values of memory locations modified by the current transaction. It extends a directory protocol in order to perform conflict detection of evicted blocks by using *sticky states*. LogTM-SE (*Signature Edition*) is a refined version of LogTM in which R/W sets are tracked using hash signatures. We use LogTM's basic algorithm to detect potential deadlocks using timestamps: A processor sets a bit if it nacks an older transaction; if in turn it receives a nack from an older transaction, this represents a potential cycle and the transaction aborts. The abort traps to a software handler, which walks the transaction log and restores the old values into memory. The system uses randomized linear backoff to reduce contention after an abort.

4.2 Simulation Methodology and Environment

We use a full-system execution-driven simulation based on the Wisconsin GEMS toolset [7], in conjunction with Virtutech Simics [6]. We use an implementation of the LogTM-SE protocol and the detailed timing model for the memory subsystem of GEMS v2.1, with the Simics in-order processor model. Simics provides functional correctness for the SPARC ISA and boots an unmodified Solaris 10.

We perform our characterization on a tiled CMP system, as described in Table 1. We use a 16-core configuration with private L1 I&D caches and a shared, multibanked L2 cache consisting of 16 banks of 512KB each. The L1 caches maintain inclusion with the L2. The cores and L2 cache banks are connected through a 2D mesh network. The private L1 data caches are kept coherent through an on-chip directory (at L2 cache banks), which maintains a bit vector of sharers and implements the MESI protocol. We compare our proposal against an ideal

Table 1. System parameters

MESI Directory-based CMP	
Core Settings	
Cores	16, single issue, in-order, non-memory IPC=1
Memory and Directory Settings	
L1 I&D caches	Private, 32KB, split, 2-way, 1-cycle latency
L2 cache	Shared, 8MB, unified, 4-way, 12 cycle-latency
L2 Directory	Full bit vector, 6-cycle latency
Memory	4GB, 300-cycle latency
Network Settings	
Topology	2D Mesh (4x4)
Link latency	1 cycle
Link bandwidth	40 bytes/cycle

Table 2. Benchmarks and inputs

Benchmark	Input	Benchmark	Input
DELAUNAY	Mesh gen3.2, min. angle 30	BARNES	4096 bodies
GENOME	8K segments, gene length 256, segment length 16	CHOLESKY	tk14
BAYES	32 variables, 1K records, 2 parents, 20%chance	RAYTRACE	teapot
KMEANS	16/16 clusters, thres. 0.05, 2048 16-dim points		
VACATION	64K entries, 4K tasks, 8 queries, 10 rel, 80 users		

implementation of LogTM-SE in which conflict detection uses *perfect* signatures – mere lists of addresses read/written by the transaction – instead of actual hash signatures that lead to unnecessary conflicts as a result of false positives.

For the evaluation, we use five transactional benchmarks extracted from the STAMP suite [3]. These benchmarks use coarse-grain transactions to execute concurrent tasks on irregular data structures such as graphs or trees. We have also selected a few *non-transactional* workloads from the SPLASH-2 suite [11], in order to evaluate our proposal with substantially different applications. Note that the latter may not be representative of future transactional applications, and are just included for comparison purposes.

4.3 Results

Figures 2 and 3 summarize the performance evaluation of the proposed directory-based conflict detection (DirCD) mechanism. We can observe how the optimized version of our proposal (DirCD+NoL1Flush) outperforms LogTM-SE in every STAMP transactional benchmark as well as in raytrace, and obtains similar results in non-transactional applications from SPLASH such as barnes or cholesky.

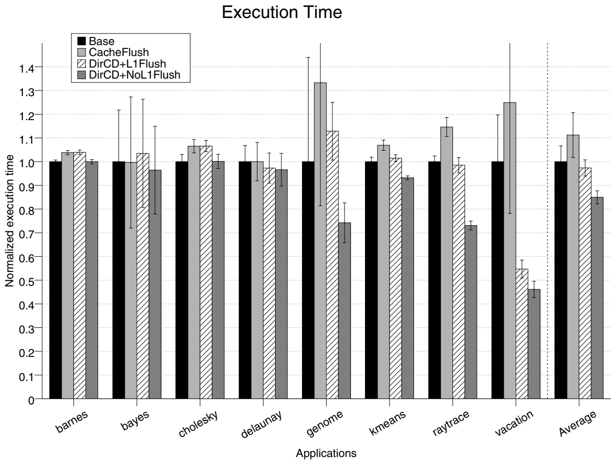


Fig. 2. Normalized execution time

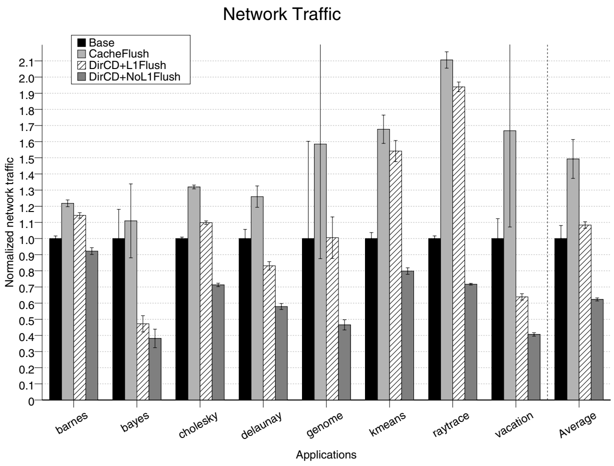


Fig. 3. Normalized network traffic

Table 3. L1 Miss Rate. Committed vs. Aborted Transactions

	L1MissRate				Commits	Aborts			
	Base	Flush	DirCD	DirCD		Base	Flush	DirCD	DirCD
			Flush	NoFlush				Flush	NoFlush
barnes	1,44%	1,71%	1,74%	1,46%	17399	316	333	306	296
bayes	1,93%	2,40%	3,67%	3,73%	526	1315	1368	1363	1285
cholesky	0,87%	1,25%	1,31%	0,87%	6567	73	41	39	75
delaunay	4,26%	6,13%	9,32%	7,66%	6312	16462	15491	15137	15408
genome	3,15%	4,49%	7,75%	2,88%	5234	3178	3908	2941	1120
kmeans	0,54%	1,08%	1,14%	0,53%	8238	6883	8172	6059	2693
raytrace	1,88%	5,14%	6,13%	2,70%	47766	203968	174393	154819	171681
vacation	4,14%	7,42%	9,32%	6,61%	4096	10573	10596	3233	2953

The performance gain of DirCD+NoL1Flush is considerable for genome (25%), raytrace (27%) and vacation (55%), three benchmarks that suffer many conflicts, as shown in the last four columns of Table 3. Other transactional workloads such as bayes, delaunay or kmeans present more modest improvements in their execution time of 3 to 7%.

First, we start by analyzing the performance degradation caused by flushing the L1 cache on every transaction, shown by the CacheFlush bar in Figure 2. This will help us understand the obtained results for our flush-based detection scheme (DirCD+Flush). As expected, flushing causes an increase in the L1 miss rate (shown in Table 3) that has a direct effect over the execution time of all benchmarks, particularly for raytrace, vacation and genome (up to 15-33%). The case of raytrace is clear: it executes a very high number of transactions (see "Commit" column of Table 3), most of which have a very small size (basically read&increment a ray id), and thus flushing the L1 on each transaction continuously leaves the post-transactional code with an almost empty data cache. The increase in on-chip network traffic (see Figure 3, is more dramatic for raytrace

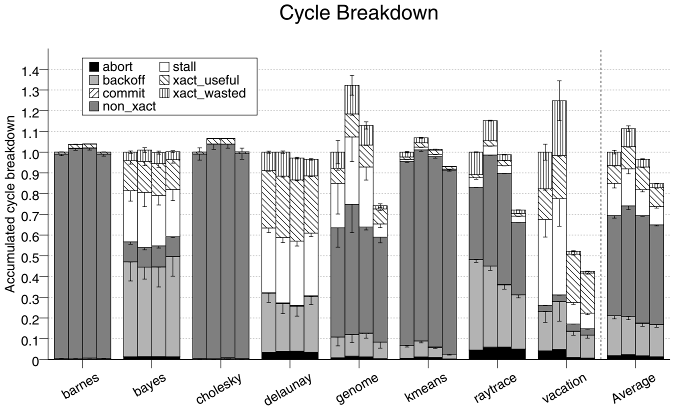


Fig. 4. Normalized cycle breakdown

than for any other benchmark, as a result of its fine-grain, abundant synchronization. However, the effects on execution time and network traffic are different for genome, a benchmark that spends the majority of its runtime in transactional code (see Figure 4). In this case, the degradation does not come from non-transactional cache misses, but from an increased number of aborted transactions – see Table 3 – that arises as a result of having transactions that span a longer period of time (probability of conflict is directly proportional to the duration).

By avoiding the flush, the DirCD+NoL1Flush configuration offers a clearer look upon the benefits of fast conflict detection than the flush-based DirCD version, as the latter introduces overheads that shadow the potential gains of our proposal over the base LogTM-SE system. The remarkable speedup achieved by genome, raytrace and vacation is not directly caused by faster conflict detection, but it happens as a result of it. As shown in Table 3 (columns "Abort"), our faster detection scheme manages to reduce the total number of aborted transactions for many benchmarks, and the gains are higher for those applications in which conflicts are not a rare event. In vacation and genome, respectively, 75% and 65% of the aborted transactions are avoided by our DirCD+NoL1Flush scheme, in comparison to the base LogTM-SE configuration. This is due to the early detection and resolution of contended situations achieved by our approach.

In the LogTM system, when a conflict is detected, the requesting processor stalls. If the conflict is detected sooner, as in the proposed scheme, the stall will likely last longer. Since execution time is determined by how quickly the system serializes conflicting transactions, detecting conflicts quicker does not speed up the execution when stalling the conflicting transaction(s) is enough to solve the conflict. However, our DirCD configuration does achieve a faster serialization of multiple conflicting transactions, when multiple conflicts cannot be resolved by stalling, but they require some transaction(s) to be aborted. In this case, the sooner the system detects the conflict, the faster it can take action and abort the appropriate transactions. The directory not only is able to detect the conflict in one hop, but it can also take action without having to wait until the conflicting block is in a base state (unlike the base approach that relies on forwarded coherence traffic), contributing to even faster detection/action. Aborting conflicting transactions earlier reduces the effect of pathological execution patterns such as *futile stall* and other conflicting interactions that affect eager CD systems like LogTM-SE [2]. The remarkable reductions in the execution time of vacation and genome are due to this quicker and more effective response. Compared to the base case, our DirCD scheme causes more aborts at first, but later on this allows more transactions to execute concurrently without interference and commit, reducing the overall number of aborts as well as the total stalled time and wasted work (see Figure 4). Quantitatively speaking, we observed this behaviour by taking a look at the first two million cycles of execution of vacation, given the two configurations (Base and DirCD+NoL1Flush) and the same random seed: we found that the former manages to commit 108 transactions by aborting 748 in that period of time, while the latter commits more than twice as many (236) at the cost of aborting around 50% more (1135). The same kind of pattern is found in other simulations with different seeds for the same benchmark.

5 Conclusions and Future Work

In this paper, we present a new approach to conflict detection targeted to TM systems built over a tiled CMP architecture. For these systems, we believe the directory constitutes a natural location for this basic transactional mechanism, and claim that extending its role to include such functionality is a natural evolution of its responsibilities within a cache coherent TM system. We propose a novel book-keeping scheme that augments each directory entry with transaction serial numbers, and describe how the detection is carried out with little assistance from the cores. The results show how the fast conflict detection achieved by our design reduces the number of aborted transactions in workloads that suffer frequent conflicts, resulting in average reductions of 15% in execution time for the selected benchmarks.

Bringing together two independent mechanisms like cache coherence and conflict detection creates a synergistic relationship that opens up a wide spectrum of new opportunities within the TM system. When combined onto the same hardware logic, both entities can cooperate *symbiotically* and accomplish new functionalities that cannot be achieved otherwise. For example, by giving the directory control over the outcome of a transaction, speculation can be applied in a variety of ways, for example, to continue the transactional execution past the occurrence of conflicting accesses. Our conflict detection mechanism already provides a commit_request/commit_ack message exchange, and could naturally support a commit_deny message that forces a transaction to abort if the speculation failed.

Acknowledgements. This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03', as well as by the EU FP6 NoE HiPEAC IST-004408. Rubén Titos is supported by a research grant from the Spanish MEC under the FPU National Plan (AP2006-04152). The authors would like to thank the anonymous reviewers for their helpful insights.

References

1. Ananian, C.S., et al.: Unbounded transactional memory. In: Proc. of the 11th Int'l. Symposium on High-Performance Computer Architecture, pp. 316–327 (February 2005)
2. Bobba, J., Moore, K.E., Yen, L., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A.: Performance pathologies in hardware transactional memory. In: Proc. of the 34rd Annual Int'l. Symposium on Computer Architecture (June 2007)
3. Cao Minh, C., et al.: An effective hybrid transactional memory system with strong isolation guarantees. In: Proc. of the 34th Annual Int'l. Symposium on Computer Architecture (June 2007)
4. Hammond, L., et al.: Transactional memory coherence and consistency. In: Proc. of the 31st Annual Int'l. Symposium on Computer Architecture, pp. 102–113 (June 2004)
5. M. Herlihy and E. B. Moss. Transactional Memory: Architectural support for lock-free data structures. In *Proc. of the 20th Annual Int'l. Symposium on Computer Architecture*, pages 289–301, May 1993.
6. Magnusson, P.S., et al.: Simics: A full system simulation platform. IEEE Computer 35(2), 50–58 (2002)
7. Martin, M.M.K., et al.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) toolset. Computer Architecture News, 92–99 (September 2005)
8. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: Log-based transactional memory. In: Proc. of the 12th Int'l. Symposium on High-Performance Computer Architecture, pp. 254–265 (February 2006)
9. Rajwar, R., et al.: Virtualizing transactional memory. In: Proc. of the 32nd Annual Int'l. Symposium on Computer Architecture, pp. 494–505 (June 2005)
10. Titos, R., Acacio, M.E., García, J.M.: Characterization of conflicts in log-based transactional memory (LogTM). In: Proc. of the 16th Euromicro Int'l. Conference on Parallel, Distributed and Network-Based Processing, pp. 30–37 (February 2008)
11. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proc. of the 22nd Annual Int'l. Symposium on Computer Architecture, pp. 24–36 (June 1995)
12. Yen, L., et al.: LogTM-SE: Decoupling hardware transactional memory from caches. In: Proc. of the 13th Int'l. Symposium on High-Performance Computer Architecture, pp. 261–272 (February 2007)

Fault-Tolerant Cache Coherence Protocols for CMPs: Evaluation and Trade-Offs

Ricardo Fernández-Pascual[1], José M. García[1], Manuel E. Acacio[1],
and José Duato[2]

[1] Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia, 30100 Murcia, Spain
{rfernandez,jmgarcia,meacacio}@ditec.um.es
[2] Dpto. de Informática de Sistemas y Computadores
Universidad Politécnica de Valencia, 46022 Valencia, Spain
jduato@disca.upv.es

Abstract. One way of dealing with transient faults that will affect the interconnection network of future large-scale Chip Multiprocessor (CMP) systems is by extending the cache coherence protocol. Fault tolerance at the level of the cache coherence protocol has been proven to achieve very low performance overhead in absence of faults while being able to support very high fault rates. In this work, we compare two already proposed fault-tolerant cache coherence protocols in a common framework and present a new one based in the cache coherence protocol used in AMD Opteron processors. Also, we thoroughly evaluate the performance of the three protocols, show how to adjust the fault tolerance parameters of the protocols to achieve a desired level of fault tolerance and measure the overhead achieved to be able to support very high transient fault rates.

1 Introduction

The number of transistors available due to current technology trends have enabled the design of progressively more powerful chips. However, these trends have several drawbacks which need to be overcome. Notably, the complexity of designing a system which takes advantage of so many components has forced architects to think of ways to simplify the design. This way, Chip Multiprocessors [2,7] have proved to be a viable way for building newer systems by exploiting thread-level parallelism. Further, tiled CMPs [14] which are built by replicating several *tiles* comprised by a core, private cache, part of a shared cache and an interconnection network interface further help in keeping complexity manageable, scale in a power-efficient way to larger number of cores and support a family of products with a varying number of tiles.

A main drawback of these trends is that, due to the miniaturization and the lower voltages, the susceptibility of future chips to transient failures will increase. Transient failures [11], also known as soft errors, occur when a component produces an erroneous output but continues working correctly after the event. Any event which upsets the stored or communicated charge can cause soft

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 555–568, 2008.

errors. Typical causes include alpha-particles strikes, cosmic rays, radiation from radioactive atoms which exist in trace amounts in all materials, and electrical sources like power supply noise or radiation from lightning.

The increased importance of transient failures means that fault-tolerance measures have to be considered across all levels of chip design. Even for commodity systems, reliability needs to be above a certain level for the system to be useful for anything. In fact, since the number of components in a chip increases and the reliability of each component decreases, it is no longer economical to design and test assuming a worst case scenario for new chips. Instead, new designs will target the common case and assume a certain rate of transient failures.

One of the components which will be affected by transient failures in a CMP is the interconnection network (IN). The IN occupies a significant part of the chip real estate and is critical to the performance of the system. It handles the communication between the cores and caches, which is done by means of a cache coherence protocol. This requires very small and frequent messages. Hence, to achieve good performance the IN must provide very low latency and should avoid acknowledgment messages and other flow-control messages as much as possible.

Fault tolerance in the IN can be provided at the network level. There are several recent proposals [3,12,13] exploring this approach. Ensuring the reliable transmission of all messages imposes significant overheads in latency, power consumption and area. In contrast, we propose to deal with transient errors in the IN at the level of the cache coherence protocol. This allows for more flexibility to design a high-performance on-chip network which can be unreliable. At the same time, the higher level information available to the coherence protocol enables it to achieve fault tolerance but avoids using acknowledgment messages in most cases, protecting only those messages which are critical for correctness. These few acknowledgments are sent out of the critical path of coherence transactions to minimize the effect of fault tolerance on performance.

We have already proposed two fault-tolerant cache coherence protocols which are described in detail in previous works [5,6]. The contributions of this paper are: an explanation of these protocols under a common framework, a description of another fault tolerant protocol which is based on a modern coherence protocol widely used in commercial systems [1] and an evaluation and comparison of this and our two previous fault tolerant protocols to show how the overhead introduced by fault-tolerance varies depending on the base protocol.

The rest of this paper is organized as follows. Section 2.1 describes the base architecture which is being extended. Section 2.2 summarizes our previously proposed fault tolerant protocols while section 3 describes a new fault tolerant protocol. The evaluation is presented in section 4 and section 5 concludes.

2 Background

2.1 Base Architecture

We assume single CMP systems built using a number of tiles [14]. Each tile contains a processor core, private L1 data and instruction caches, a bank of the logically shared L2 cache and a network interface. The L2 cache is logically shared by all cores but it is physically distributed among the tiles. Each tile has its network interface which connects it to the on-chip IN. We assume in-order processors since that seems the most reasonable approach to build power-efficient CMPs with many cores. While we have assumed a tiled architecture and in-order processors, these choices are not constraints of the evaluated coherence protocols, whose functionality and correctness is not affected if out-of-order cores are used or a different arrangement is used instead of tiles.

We consider two base architectures: one using a token-based cache coherence protocol (TokenCMP) [10] and another one using a more traditional directory-based protocol (DirCMP). A third base architecture is described in section 3.

TokenCMP is a protocol based on token coherence which targets multiple CMPs and is well suited for single CMPs. Token coherence provides a framework for defining coherence protocols by separating the definition in a correctness substrate and a performance policy which define how the nodes exchange a fixed number of tokens among them. Most requests are *transient requests* which, in the case of *TokenCMP*, are broadcasted to all other nodes without ordering guarantees and without even a guarantee of being satisfied. *Token counting* rules ensure that coherency is maintained while *persistent requests* ensure forward progress by providing serialization when races between transient requests are detected. TokenCMP uses a performance policy similar to TokenB (*Token-using-broadcast*) with distributed arbitration for persistent requests.

DirCMP is a traditional MOESI-based directory cache coherence protocol [4] which uses an on-chip directory to maintain coherence between several private L1 caches and a shared non-inclusive L2 cache. It uses a directory cache in L2 and the L2 effectively acts as the directory for the L1 caches.

2.2 Our Previous Work

We have designed two fault-tolerant cache coherence protocols for CMPs based on two different approaches to cache coherence: token coherence and directory coherence. Both protocols have been shown to provide fault-tolerance with respect to transient faults in the IN with very little overhead. FtTokenCMP [5] is a token based coherence protocol which extends TokenCMP with fault tolerance, while FtDirCMP [6] is another fault-tolerant coherence protocol which is based in a more traditional directory protocol which we call DirCMP.

The fault tolerance measures of both protocols are similar in their intent and functionality and differ mostly in the implementation. In our experience, a fault tolerant cache coherence protocol needs to provide the following things: a fault detection mechanism, a fault recovery mechanism, and a mechanism to ensure that data is never lost or corrupted. Both protocols rely on error detection codes in messages to discard corrupted messages. It is assumed that the error detection code checks the whole message. Thus, from the point of view of the coherence protocols, a message can either arrive correctly or not arrive at all.

In both protocols, fault detection is achieved by means of a number of timeouts which detect deadlocks caused by discarded messages. This fault detection

Table 1. Timeouts summary for FtDirCMP and FtTokenCMP

Timeout	When is it activated?	When is it deactivated?	Action when it triggers?
FtDirCMP			
Lost Request	When a request is issued.	When the request is satisfied.	The request is reissued with a new serial number.
Lost Unblock	When a request is answered.	When the unblock message is received.	An *UnblockPing* is sent.
FtTokenCMP			
Lost Token	When a persistent request becomes active.	When the persistent request is deactivated.	Request a token recreation.
Lost Persistent Deactivation	When an persistent request is activated.	When the persistent request is deactivated.	Send a persistent request ping.
Both protocols			
Lost Data	When a backup state is entered.	When the Ownership Acknowledgement arrives.	Issue an *OwnershipPing* / Request a token recreation.
Lost Backup Deletion Ack.	When a line enters the blocked state.	When the Backup Deletion Acknowledgement arrives.	Reissue the *AckO* / Request a token recreation.

mechanism is reliable and valid for every coherence protocol where a discarded message can be either harmless or lead to a deadlock in the same or a subsequent memory transaction. This is the case of TokenCMP, where discarded transient requests are harmless and the rest of message types lead to deadlock; and in the case of DirCMP where every discarded message leads to a deadlock. However, not all cache coherence protocols have this property: for example, some protocols do not require acknowledgments for invalidation messages, hence discarding an invalidation message would lead to an incoherence instead of a deadlock. Table 1 shows a summary of the timeouts used by each protocol.

Also, both protocols use essentially the same mechanism to avoid data loss, ensuring reliable transmission of owned data by means of exchanging a pair of acknowledgments. The mechanism works as follows: when a cache sends owned data to another cache, it keeps a backup copy of it. This backup copy may be used by the respective recovery mechanism if necessary, but it cannot be used by the cache for any other purpose. The backup will be kept until an *ownership acknowledgment* sent by the receiver arrives. On the other hand, the cache which receives the data can use it as soon as it arrives, but it cannot send it to another cache until it receives a *backup deletion acknowledgment* sent by the previous owner once its backup has been discarded. The last restriction is necessary to ensure that there is no more than one backup copy of each cache line, because otherwise fault recovery would be significantly more complex.

These acknowledgments are sent out of the critical path of cache misses so they do not directly affect the execution time of programs. Also, in many cases the acknowledgments are piggybacked in other messages of the same coherence transaction. However, the increased network traffic caused by this mechanism is the main overhead incurred by the fault tolerant measures of both protocols.

The fault recovery mechanism is different for each protocol. In FtTokenCMP, fault recovery is achieved by means of a centralized mechanism called the *token recreation process* arbitrated by the memory controller. This process works as long as there is a valid copy of data in some cache or one and only one backup copy (which is guaranteed by the owned data transmission mechanism described above). The memory controller attends token recreation requests in FIFO order to avoid livelock and it works sending messages to every cache asking it to invalidate all tokens and send back to memory any data that it may have. Once the memory receives the data or invalidation acknowledgments from every cache, it sends it to the cache which requested the recovery with a new set of tokens.

To avoid creating an incoherence due to stale response messages still traveling through the IN after a *token recreation*, all coherence responses are tagged with a *token serial number* (TSN) which is increased during the token recreation process. Messages with a wrong TSN are discarded when received by any node. Token serial numbers are stored in every node in a dedicated structure (the *TSN table*), but only for those cache lines which have a serial number different than 0. We have found that having a very small number of entries of only a few bits each is enough for good results. When all entries are used, one of them is evicted setting its serial number to 0 by means of the *token recreation process*.

In contrast, FtDirCMP achieves fault recovery reissuing requests with a different *request serial number*. FtDirCMP does not need an specific serialization point for fault recovery since the directory (or the on-chip L2 directory cache) acts already as the serialization point for all requests. These reissued requests need to be identified as such by the node that answers to them and not be treated like an ordinary request. In particular, a reissued request should not wait in the incoming request buffer to be attended by the L2 or the memory controller until a previous request is satisfied, because that previous request may be precisely the older instance of the request that is being reissued in case of a false positive.

Since stale responses to a few reissued request messages may lead to an incoherence in FtDirCMP, we use *request serial numbers* to discard responses which arrive too late (when the request has already been reissued). Every message carries a serial number. Request serial numbers are chosen by the cache that issues the request while responses or forwarded requests will carry that of the request that they are answering to. When a request is reissued, it will be assigned a new serial number which will allow to distinguish between responses to the old request and to the new one. Nodes must remember the serial number of the requests that they are currently handling and discard any message which arrives with an unexpected serial number or from an unexpected sender. This information needs to be updated when a reissued request arrives.

In some cases, both protocols achieve deadlock recovery issuing ping messages when a timeout triggers to force the reissue of a message which is expected to finish a coherence transaction, like an *Unblock* message in case of FtDirCMP or a *Persistent Request Deactivation* message in case of FtTokenCMP.

The *token serial numbers* used in FtTokenCMP serve a similar purpose to *request serial numbers* used in FtDirCMP (e.g.: being able to discard stale messages after fault recovery which could cause an incoherence), but the latter are easier to implement and more scalable. *Token serial numbers* are associated with each cache line and need to be updated in a coordinated fashion during the *token recreation process*. Hence, they required an additional structure in each cache to store them (only for those hopefully few lines that had a token serial number different than 0, but even for lines which were not currently in any cache). On the other hand, *request serial numbers* are associated with individual requests and so they are short-lived information which can be stored in the MSHR. However, *token serial numbers* do not need to be carried in request messages (only in responses) while *request serial numbers* are sent with every request and need to be propagated with every message which is sent as consequence of the request.

Notice that discarding any message in FtDirCMP or FtTokenCMP is always safe (even if it could be not strictly necessary in some cases) since the protocol already has provisions for lost messages of any type.

3 A New Broadcast-Based Cache Coherence Protocol

No real system has been implemented yet using a coherence protocol based on the token framework. Also, many cache coherence protocols which are used in widely used systems cannot be precisely categorized as snoopy-based nor directory-based. AMD Hammer [1] is one of these protocols. It targets systems with a small number of processors using a tightly-coupled point-to-point unordered IN.

In this work, we have implemented HammerCMP which is an adaptation of AMD Hammer protocol to the tiled CMP environment and we have used it as a base for FtHammerCMP, a new fault tolerant protocol for small scale CMPs.

Like DirCMP, HammerCMP sends requests to a home L2 bank which acts as the serialization point for requests to its cache lines. There is no directory information, and all requests are forwarded using broadcast to all other caches. All of them answer to the forwarded requests sending either an acknowledgment or a data message to the requestor. When the requestor receives all the acknowledgments informs to the home L2 controller that the miss has been satisfied.

HammerCMP avoids the overhead of directory information and the latency of accessing the directory structure at the cost of much more IN traffic. Also, all processors need to intervene in all misses, like in a snoopy protocol.

Using the principles described in section 2.2, FtHammerCMP adds fault tolerance measures to HammerCMP as the ones described for FtDirCMP. It uses the same set of timeouts for detecting faults and reissues requests using different request serial numbers in a way very similar to FtDirCMP for recovering from faults. Reliable owned data transference is done using the same pair of acknowledgments as the other two protocols.

4 Evaluation

4.1 Methodology

We have performed full system simulations of a mix of scientific applications with fault injection with the aims of determining adequate values for some protocol parameters, assess the fault tolerance capability of each protocol and measure the overhead introduced by the fault tolerance measures. For this, we have used a custom version of Multifacet GEMS [9] detailed memory model and Virtutech

Table 2. Characteristics of simulated architectures and input sizes used for benchmarks in the simulations

(a) System characteristics

16-Way Tiled CMP System	
Processor speed	2 GHz
Cache parameters	
Cache line size	64 bytes
L1 cache:	
Size, associativity	32 KB, 4 ways
Hit time	2 cycles
Shared L2 cache:	
Size, associativity	1024 KB, 4 ways
Hit time	15 cycles
Memory parameters	
Memory access time	300 cycles
Memory interleaving	4-way
Network parameters	
Topology	2D Mesh
Non-data message size	8 bytes
Data message size	72 bytes
Channel bandwidth	64 GB/s

(b) Input sizes

Benchmark	**Input Size**
Barnes	8192 bodies, 4 time steps
Cholesky	tk16.O
FFT	256K complex doubles
Ocean	258 × 258 ocean
Radix	1M keys, 1024 radix
Raytrace	10Mb, teapot.env scene
Tomcatv	256 points, 5 iterations
Unstructured	Mesh.2K, 5 time steps
Water-NSQ	512 molecules, 4 time steps
Water-SP	512 molecules, 4 time steps

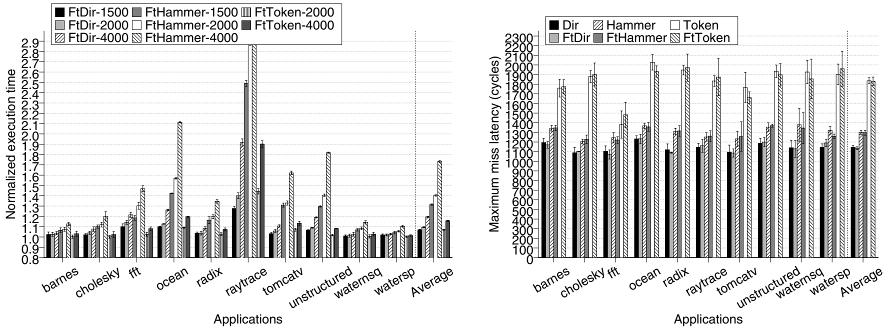
Simics [8]. Every simulation has been performed several times using different random seeds to account for the variability of multithreaded execution, this is represented by the error bars in the figures which enclose the resulting 95% confidence interval of the results. We have simulated tiled CMP systems as described in section 2.1. Table 2(a) shows the most relevant parameters of the systems.

Finally, we have used a selection of scientific applications for the evaluation: Barnes, Cholesky, FFT, Ocean, Radix, Raytrace, Water-NSQ, and Water-SP are from the SPLASH-2 benchmark suite. Tomcatv is a parallel version of a SPEC benchmark and Unstructured is a computational fluid dynamics application. The experimental results reported here correspond to the parallel phase of each program only. Problem sizes are shown in table 2(b).

4.2 Adjusting the Fault Detection Timeouts

All fault tolerant protocols achieve fault detection by means of a number of timeouts. Each protocol requires up to four timeouts which are active at different places and times during a memory transaction or cache replacement. The value of these timeouts determine the latency of fault detection, hence shorter values help to achieve lesser performance degradation in presence of faults since fault recovery will start earlier. For example, for the three fault tolerant protocols considered in this work figure 1(a) shows how the execution time increases with the value of these timeouts under a fixed fault rate.

Since false positives occur when a timeout triggers before a miss has had enough time to be satisfied, to avoid false positives the timeout values should be large enough to allow every memory transaction to finish, assuming that no fault occurs. Figure 1(b) shows the measured maximum latency in CPU cycles of each protocol when no faults occur and disabling all the timeouts.



(a) Execution time varying timeouts

(b) Maximum miss latency

Fig. 1. Relative execution time with respect to DirCMP without faults for each fault tolerant protocol with 250 corrupted messages per million using different values for the fault detection timeouts and maximum miss latency (in cycles) of each coherence protocol without faults

Looking at figure 1(b), we can see that the maximum latency of the fault-tolerant protocols is almost exactly the same than that of their corresponding non fault-tolerant counterpart. This is expected, since the behavior of the fault-tolerant protocols when no timeout triggers is almost the same than that of the non fault-tolerant ones, except for the ownership acknowledgments which are sent out of the critical path of cache misses.

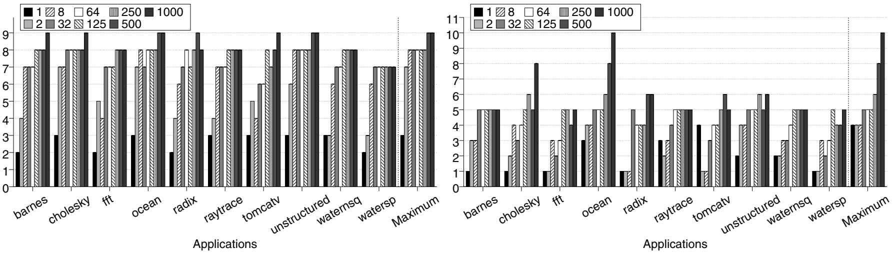
This latency is around 1250 cycles for the FtDirCMP protocol, 1350 for the FtHammerCMP protocol and 2000 for the FtTokenCMP protocol. Hence, we can choose any value greater than those for the timeouts to avoid having any false positive for these workloads. Using shorter values is still possible but would increase the number of false positives and could degrade performance and increase network traffic due to the retried requests or token recreation requests. However, if the chosen values are too low (lower than the time required to finish a transaction), the recovery mechanism would be invoked too frequently preventing forward progress.

Finally, we have considered using different values for each of the four timeouts of each protocol, but our experiments do not show any significant advantage in doing so.

We have chosen a value of 2000 cycles for all timeouts in the FtTokenCMP protocol and 1500 cycles in the FtDirCMP and FtHammerCMP protocols. These values are large enough to avoid false positives in every case and, as shown below, achieve very low performance degradation when faults actually occur. Making this value smaller achieves very little benefit while significantly increasing the risk of false positives.

4.3 Effect of the Request Serial Number Size in Fault-Tolerance

In the case of FtDirCMP and FtHammerCMP, the ability to correctly recover from faults depends on the number of bits used for encoding the request



(a) FtDirCMP

(b) FtHammerCMP

Fig. 2. Required RSN bits to discard every old response to a reissued message

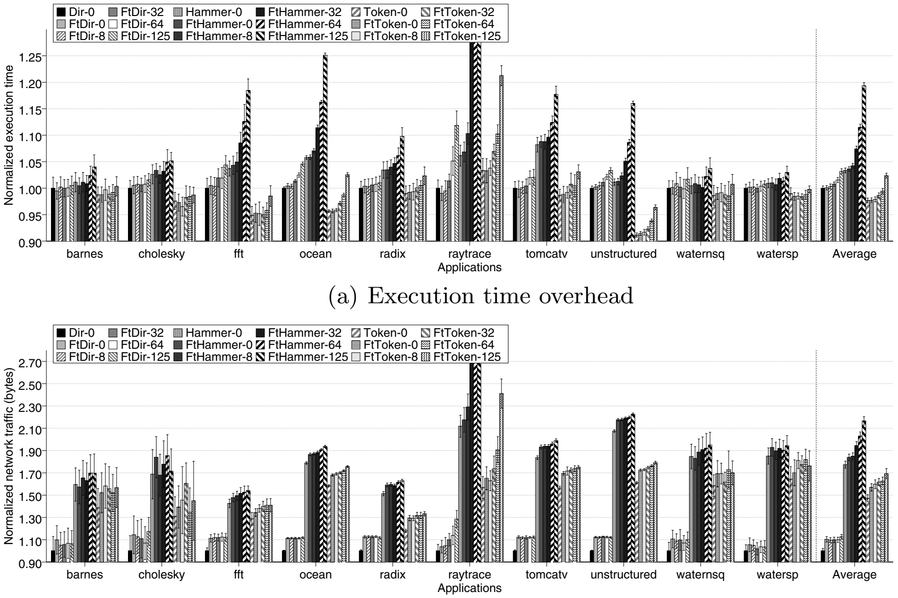
serial number which is used to discard stale responses to reissued requests (for example, to discard old acknowledgments to reissued invalidation messages which could lead to incoherence). This number should be as low as possible to reduce overhead in terms of increased message size and hardware resources to store it while being sufficient to ensure that when a request is reissued (even several times in a row) every response to the old request is discarded. Since the number of reissued messages increases as the fault rate increases, the number of bits used to encode request serial numbers determines the maximum fault rate supported.

To measure this, we have performed simulations of FtDirCMP using a wide variety of fault rates. We have used 32-bit request serial numbers for those simulations but we have recorded how many lower order bits were required to distinguish all the request serial numbers that needed to be compared. For doing this, every time that two request serial numbers are compared, we record the position of the least significant bit which is different in both numbers. Then, we assume that the maximum of all these measures is an upper bound of the number of bits required to ensure correctness for each fault rate. These results are shown in figure 2.

As it can be seen, when using the FtDirCMP protocol 9 bits are enough for all the tested fault rates and 8 bits suffice for fault rates up to 250 corrupted messages per million. In case of using the FtHammerCMP protocol, 8 bits provide fault tolerance up to 500 corrupted messages per million, while 10 bits are required for the maximum tested fault rate, 1000 corrupted messages per million. Hence, we have chosen to use 8 bits to encode the request serial numbers in the rest of our experiments for both protocols which is enough to achieve fault tolerance up to 250 corrupted messages per million, which is already an unrealistic and unreasonably high failure rate.

4.4 Execution Time Overhead

We have measured the execution time of each one of the fault-tolerant protocols using the fault tolerance parameters determined above with several message loss rates and compared it to the execution time of the non fault tolerant protocols in a fault-free scenario. The results are shown in figure 3(a). Fault rates are



(a) Execution time overhead

(b) Network overhead (bytes)

Fig. 3. Execution time and network overhead of each protocol for several fault rates

expressed in number of messages discarded per million of messages that travel trough the network and all results are normalized with respect to the execution time of the DirCMP protocol.

We can see that the run-time overhead of each fault-tolerant protocol when compared to its non fault-tolerant counterpart in a fault-free scenario is not measurable. This is consistent with the fact that, when no faults occur, the only difference in the behavior of the fault-tolerant protocols with respect to the non fault-tolerant ones is just the extra acknowledgments used to ensure reliable owned data transmission, which are sent out of the critical path of misses.

For these workloads, both FtTokenCMP and FtDirCMP achieve very similar execution times when no faults occur (less than 3% difference on average). FtHammerCMP execution time difference is less than 4% higher than FtDirCMP also, and 6% higher than FtTokenCMP.

As the fault rate increases, so does the execution time of each protocol. In the case of FtDirCMP and FtTokenCMP, the average performance degradation is almost unmeasurable until the message loss rate reaches 32 corrupted messages per million. However, even when the fault rate reaches 64 corrupted messages per million, the execution time of FtTokenCMP is lower than the execution time of the directory protocol in a fault free scenario.

On the other hand, HammerCMP and FtHammerCMP without faults have the same performance than FtDirCMP under a fault rate of 125 messages lost per million. Also FtHammerCMP performance degradation starts being significant under a fault rate of 8 messages corrupted per million and the rate at which it increases is noticeably worse than in the case of the other protocols. This is due to the much higher network traffic of HammerCMP and FtHammerCMP in comparison to all the other protocols even in absence of faults as can be seen in figure 3(b).

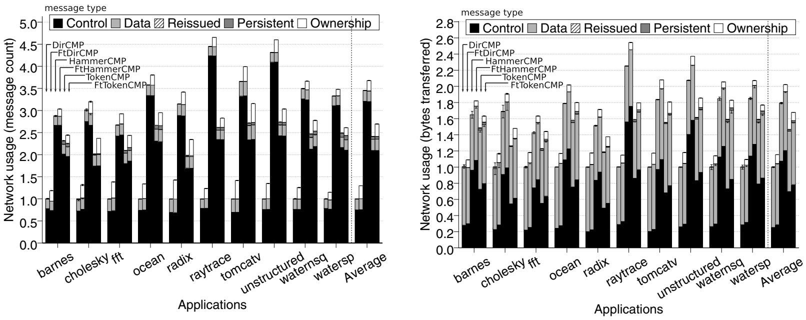
4.5 Network Overhead

In absence of faults, the most important difference in the behavior of our protocols with respect to their non fault-tolerant counterparts is the exchange of acknowledgments to ensure that owned data is transferred safely and avoid data loss. Although they are sent out of the critical path of cache misses so that they do not have effect in the miss latency, these acknowledgments introduce additional network traffic which is the main cost of the fault tolerance measures.

We have measured the network overhead of our proposal in terms of the relative increase in the number of messages and the number of bytes transmitted through the network. We have increased one byte the message sizes of the fault tolerant protocols with respect to the non fault-tolerant ones to accommodate the *request serial numbers* and *token serial numbers*. This means 1.14% increase in size for data messages and 12.5% increase for control messages. The results of these measurements are shown in figure 4.

We can see that, in terms of message traffic, the overhead of the fault-tolerant protocols comes entirely from the acknowledgments used to ensure reliable data transmission ("*Ownership*" part of each bar). This overhead is less than 30% for all our fault-tolerant protocols. Moreover, the overhead drops considerably when it is measured in terms of bytes, even considering that every message is one byte longer in the fault-tolerant protocols.

Figure 3(b) shows the network overhead under several fault rates. The network traffic increases slowly with the fault rate due to the reissued messages or the token recreation messages. In the case of FtDirCMP, the increase is



(a) Number of messages

(b) Bytes transmitted

Fig. 4. Network overhead of the fault tolerant protocols

almost unmeasurable for the fault rates shown. However, as can be seen for FtHammerCMP, once the network traffic reaches certain point (around 1.9 in our plot), the slope becomes steeper. This is due to the fact that the capacity of the network is exceeded and this increases the average latency which in turn causes a number of false positives which lead to more reissues which further increase the network traffic and consequently the execution time. Hence, network capacity can become a limiting factor for the fault tolerance of our protocols.

4.6 Hardware Requirements

The *token serial number table* is implemented with a small associative table at each cache and at the memory controller to store those serial numbers whose value is not zero. Using two bits to encode the serial number and 16 entries at each node is enough for supporting the fault rates used in this paper. If the tokens of any line need to be recreated more than 4 times the counter wraps to zero (effectively freeing an entry in the table) and if more than 16 different lines need to be stored in the table, the least recently modified line is evicted by means of using the token recreation process to set its serial number to zero.

On the other hand, *request serial numbers* do not need to be kept once the memory transaction has been completed. They can be stored in the MSHR or (optionally) in a small associative structure in cases where a full MSHR is not needed. As shown in section 4.3, using 8 bits to encode request serial numbers is enough to achieve tolerance to very high fault rates, and even less bits are required to support more realistic but still very high fault rates.

Also, to be able to detect reissued requests in FtDirCMP and FtHammerCMP, the identity of the requester currently being serviced by the L2 or the memory controller needs to be recorded, as well as the identity of the receiver of owned data when transferring ownership from one L1 cache to another to be able to detect reissued forwarded requests.

The timeouts used for fault detection require the addition of counters to the MSHRs or a separate pool of timeout counters. Although there are up to four different timeouts involved in any coherence transaction, no more than one counter is required at any time in the same node for a single coherence transaction. In the case of FtTokenCMP, all but one timeout can be implemented using the same hardware already used to implement the starvation timeout required by token protocols. Also, our fault-tolerant protocols require one extra virtual channel than their non fault-tolerant counterparts.

Finally, a less important source of overhead is the increased pressure in caches and writeback buffers because of the blocked ownership and backup states and the effect of the reliable ownership transference mechanism in replacements. When a backup buffer or a writeback buffer is used, we have not been able to detect any effect in the execution time due to these reasons. The size of the writeback buffer may need to be increased, but our previous work [5] shows that one extra entry would be enough to avoid any slowdown.

5 Conclusions

We propose implementing fault tolerance measures at the cache coherence protocol level to deal with transient faults in the interconnection network of CMPs and provide several cache coherence protocols which can ensure the correct execution of parallel programs using a non reliable on-chip IN.

In this work, we have presented a new fault tolerant protocol based on AMD Hammer protocol which could be useful for small scale CMPs. We have thoroughly compared and evaluated the performance of two previously presented fault tolerant cache coherence protocols and the new one. We have shown that the overhead imposed in the execution time due to the fault tolerant measures is negligible. Further, we have shown that the performance impact of moderate fault rates in the IN is insignificant when using our protocols.

We have explained how to tune the fault tolerance parameters of the protocols to achieve the desired level of fault tolerance, performance degradation in presence of faults and overhead in absence of faults. We have shown that, even for fault rates which are unrealistically high, the hardware overhead of our proposals is low. The main cost of our fault tolerance measures is a moderate increase in network traffic, but this increase is much lower that the difference in network usage between protocols, specially considering currently used protocols like AMD Hammer.

Our evaluation shows that a token coherence based protocol can provide sightly better performance than a directory based one even when the token based protocol is subjected to higher fault rates, but at the cost of much higher network usage. We have found that the network usage of our protocols increases with the fault rate and hence network capacity can be a limiting factor for fault tolerance. Due to the efficient network usage of directory-based protocols and the small difference in performance with respect to the other two fault tolerant protocols shown in our evaluation, we think that FtDirCMP is a good cache coherence protocol for large scale tiled CMPs.

Acknowledgements

This work has been jointly supported by the Spanish MEC and European Comission FEDER funds under grants “Consolider Ingenio-2010 CSD2006-00046” and “TIN2006-15516-C04-03”, and also by the EU FP6 NoE HiPEAC IST-004408.

References

1. Ahmed, A., Conway, P., Hughes, B., Weber, F.: AMD Opteron TM Shared-Memory MP Systems. In: 14th HotChips Symp. (August 2002)
2. Barroso, L.A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., Verghese, B.: Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In: Proc. of 27th Int'l. Symp. on Computer Architecture (ISCA 2000), pp. 282–293 (June 2000)

3. Constantinides, K., Plaza, S., Blome, J., Zhang, B., Bertacco, V., Mahlke, S., Austin, T., Orshansky, M.: BulletProof: a defect-tolerant CMP switch architecture. In: 12th Int'l. Symp. on High-Performance Computer Architecture (HPCA 2006), pp. 3–14 (February 2006)
4. Culler, D.J., Singh, J.P., Gupta, A.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers, San Francisco (1999)
5. Fernndez-Pascual, R., García, J.M., Acacio, M.E., Duato, J.: A low overhead fault tolerant coherence protocol for CMP architectures. In: 13th Int'l. Symposium on High-Performance Computer Architecture (HPCA 2007), pp. 157–168 (February 2007)
6. Fernández-Pascual, R., García, J.M., Acacio, M.E., Duato, J.: A fault-tolerant directory-based cache coherence protocol for shared-memory architectures. In: The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2008) (June 2008)
7. Hammond, L., Hubbert, B.A., Siu, M., Prabhu, M.K., Chen, M., Olukotun, K.: The Stanford Hydra CMP. IEEE MICRO Magazine 20(2), 71–84 (2000)
8. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer 35(2), 50–58 (2002)
9. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. Computer Architecture News 33(4), 92–99 (2005)
10. Marty, M.R., Bingham, J.D., Hill, M.D., Hu, A.J., Martin, M.M.K., Wood, D.A.: Improving multiple-CMP systems using token coherence. In: 11th Int'l Symposium on High-Performance Computer Architecture (HPCA 2005), pp. 328–339. IEEE Computer Society, Los Alamitos (2005)
11. Mukherjee, S.S., Emer, J., Reinhardt, S.K.: The soft error problem: An architectural perspective. In: 11th Int'l Symposium on High-Performance Computer Architecture (HPCA 2005) (February 2005)
12. Murali, S., Theocharides, T., Vijaykrishnan, N., Irwin, M.J., Benini, L., De Micheli, G.: Analysis of error recovery schemes for networks on chips. IEEE Design and Test of Computers 22(5), 434–442 (2005)
13. Park, D., Nicopoulos, C., Kim, J., Vijaykrishnan, N., Das, C.R.: Exploring fault-tolerant network-on-chip architectures. In: Procs. of the 2006 Int'l Conf. on Dependable Systems and Networks (DSN 2006), pp. 93–104 (2006)
14. Taylor, M.B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H.: The raw microprocessor: A computational fabric for software circuits and general purpose programs. IEEE Micro 22(2), 25–35 (2002)

SDRM: Simultaneous Determination of Regions and Function-to-Region Mapping for Scratchpad Memories[⋆]

Amit Pabalkar, Aviral Shrivastava, Arun Kannan, and Jongeun Lee

Department of Computer Science and Engineering,
Arizona State University, Tempe, AZ 85281
{amit.pabalkar,aviral.shrivastava,
arun.kannan,jongeun.lee}@asu.edu

Abstract. Many programmable embedded systems feature low power processors coupled with fast compiler controlled on-chip scratchpad memories (SPMs) to reduce their energy consumption. SPMs are more efficient than caches in terms of energy consumption, performance, area and timing predictability. However, unlike caches SPMs need explicit management by software, the quality of which can impact the performance of SPM based systems. In this paper, we present a fully-automated, dynamic code overlaying technique for SPMs based on pure static analysis. Static analysis is less restrictive than profiling and can be easily extended to general compiler framework where the time consuming and expensive task of profiling may not be feasible. The SPM code mapping problem is harder than bin packing problem, which is NP-complete. Therefore we formulate the SPM code mapping as a binary integer linear programming problem and also propose a heuristic, determining simultaneously the region (bin) sizes as well as the function-to-region mapping. To the best of our knowledge, this is the first heuristic which simultaneously solves the interdependent problems of region size determination and the function-to-region mapping. We evaluate our approach for a set of MiBench applications on a horizontally split I-cache and SPM architecture (HSA). Compared to a cache-only architecture (COA), the HSA gives an average energy reduction of 35%, with minimal performance degradation. For the HSA, we also compare the energy results from our proposed SDRM heuristic against a previous static analysis based mapping heuristic and observe an average 27% energy reduction.

Keywords: Compilers, Code overlay, Static code analysis, Scratchpad memory.

1 Introduction

The first generation embedded systems were limited to fixed, single functionality devices like digital watches, calculators, coffee makers etc. Modern embedded systems have evolved into programmable, highly complex, multi-functionality devices including portable music players, gaming consoles, PDAs, GPSs and cellular phones. These systems must exhibit high performance while at the same time consume less power,

[⋆] This work was partially funded by grants from Microsoft, Raytheon and Startdust Foundation.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 569–582, 2008.

as they operate on battery. Design of such systems thus becomes extremely challenging due to multi-dimensional and stringent design constraints.

Modern embedded processors increase performance by employing memory hierarchies consisting of caches or scratchpads or both. Caches improve performance by exploiting the spatial and temporal locality in the application, without any changes to the application itself. However, these improvements are achieved through use of tag arrays, comparators and management logic which in certain processors like StrongARM, can consume more than 40% of the total power budget [5].

Scratchpad Memories (SPM) on the other hand are devoid of power hungry tag arrays and comparators. Compared to caches they consume less energy per access and occupy smaller on-chip area. While previous works have demonstrated that a SPM may require on an average 40% less energy and 34% less die area compared to a cache of same size [3], the compiler is now responsible for managing the SPM contents. This involves inserting explicit instructions in the program to move code or data between SPM and the main memory. A good technique for mapping the program contents onto SPM thus becomes very critical for efficiently utilizing the SPM with minimal runtime transfer overhead. Since code exhibits more locality than data, mapping code should provide us with more power reduction. Therefore, in this work we focus on mapping application code onto the SPM.

Most code mapping techniques for SPMs require profiling to find the optimal mapping of applications. Profiling however, limits their applicability, not only because of the difficulty in obtaining reasonable profiles, but also due to high space and time requirements to generate a profile. Instead, in this work, we use compile time static analysis to eliminate profiling and the overhead associated with it. Our static analysis is based on a new data structure, Global Call Control Flow Graph (GCCFG), which captures the function call *sequence* as well as the control flow information like loops and conditionals. Our GCCFG can give not only the execution counts (estimated from the control flow) but also the execution sequences of functions (from control flow, call graph, and call sequence). This makes GCCFG more precise than just a call or a control flow graph in modeling the runtime behavior of an application.

Traditional approaches for SPM utilization breaks down the SPM mapping problem into two smaller problems. The first problem, termed as *memory assignment* or *'what to map'* involves partitioning the application code into SPM mapped and main memory spilled. This division eliminates code segments whose cost of transfer from memory to SPM is greater than the profit of execution from SPM. However since our architecture has a direct memory access controller, the transfer cost is negligible and it is always profitable to execute the entire code from the SPM. We therefore do not consider the 'what to map' problem in this work. The focus of this work is the second problem, termed as address assignment or 'where to map' which involves determining the addresses on the SPM where the code will be mapped.

Code mapping techniques for SPM can be classified into static and dynamic techniques. In static techniques, SPM is loaded once during program initialization occupying the entire SPM and the contents do not change during the execution of the program. This implies that the static techniques need not address the 'where to map' issue; they only solve the 'what to map' issue. The reduced utilization of SPM at runtime means less scope for energy reduction. Dynamic techniques on the other hand, replenishes the contents of the SPM with different code segments during program execution by overlaying multiple code segments. For most efficient management, the SPM can be partitioned into bins or regions and multiple code segments with non-overlapping live ranges should be mapped to different regions. Thus a dynamic technique for code mapping can be broken down into

1. Partition of the SPM into optimal number of regions
2. Overlaying the code objects onto the regions

Although previous dynamic approaches viz. first-fit [11] and best-fit [10] have proposed solutions for the second subproblem, none of the above approaches determine the optimal size and number of regions. These heuristics assume a pre-determined number of regions and may cause spilling of critical functions to the main memory. In fact, the above two sub-problems have a cyclic dependency and if solved independently one after another, the combined solution is sub-optimal. In this paper we propose a Simultaneous Determination of Region and Function-to-Region Mapping (SDRM) technique which solves the two subproblems at the same time. Regions are created as each function gets mapped to the SPM and are resized if the mapped function is greater than the existing region size, without violating the total size constraints. To compare the optimality of our technique, we also formulate a binary ILP to solve the code mapping problem. Our experiments using MiBench benchmark suite indicate that our technique can find near-optimal solutions compared to the ILP solutions and they are 27% better than the solution obtained by first-fit heuristic.

2 Related Work

As discussed in the previous section, SPM mapping techniques can be classified into static and dynamic techniques for both code and data. Papers [1, 2, 8] present static techniques for SPM allocation. While authors in [8] use a knapsack algorithm for static assignment of code and data objects, authors in [1] propose a dynamic programming approach to select and statically assign code objects to maximize energy saving. The static approach in [2] concentrates only on data objects.

While static approaches are easy to formulate, they significantly limit the scope of energy reduction. Therefore a majority of research [4,9,10,11] have focussed on solving both code and data mapping problem using dynamic techniques. In this research work, we also propose a dynamic technique, but overlay only code objects due to greater energy reduction potential.

The approach in [9] formulates a binary ILP to select an optimal set of code blocks and corresponding copy points which minimize energy consumption. However their approach does not solve the 'where to map' problem. The authors in [4] propose another dynamic technique for systems with virtual memory, where the page fault exception mechanism of MMU is used to copy code blocks to SPM on demand. However this technique dictates some hardware enhancement. On the contrary our technique is a pure software method and does not impose any architectural changes. The research in [10] proposes yet another dynamic profile SPM allocation technique where the authors give a heuristic for classification of code, stack and global data into SPM and cache, and a best-fit heuristic to solve the 'where to map' problem. However their technique use compaction to minimize fragmentation which can incur a significant overhead and can be prohibitive in embedded systems.

Except [11], which use static analysis for code objects, all the above techniques use profiling to find the execution count of objects. A relative advantage of static analysis over profiling has already been discussed in the previous section. The technique that we propose is closest to the approach presented by authors in [11]. They formulate an Integer Linear Programming (ILP) problem to partition the memory objects into SPM and main memory and then use another ILP to determine the address assignment. Since an ILP is intractable for large size programs they propose a first-fit heuristic to solve the 'where to map' problem. However, the heuristic in their work use a predetermined number and size of regions. In contrast, the technique in our work computes the number and size of regions while solving the mapping problem itself. We also formulate a binary ILP and show that our heuristic is near-optimal to the ILP solution. In the next section we formulate a generic problem definition for the mapping of code to SPM.

3 Problem Definition

INPUT:

- Global Call Control Flow Graph (GCCFG). GCCFG is an ordered directed graph D=(V_f, V_l, V_i, E), where each node $v_f \in V_f$ represents function or F-node, $v_l \in V_l$ represents a loop or L-node, $v_i \in V_i$ represents a conditional or I-node and edge $e_{i,j} \in E \ni v_i, v_j \in V_f \bigcup V_l \bigcup V_i$ is a directed edge between F-nodes, L-nodes and I-nodes. If v_i and v_j are both F-nodes, the edge represents a function call. If either one is a L-node, the edge represents a control flow. If either one is a I-node, the edge represents a conditional flow. If both are L-nodes the edge represents nested control flow. Recursive functions are represented by edges whose source and destination are the same. The edges of a node are ordered, i.e. if a node has two children, the left node is called before the right node in the control flow path of the program. Each F-node is assigned a statically determined weight w_i representing its execution count.
- Set S = $\{s_1, s_2...s_f\}$, representing the functions sizes (F-nodes V_f in the GCCFG).
- $E_{spm/access}$ and $E_{i-cache/access}$, representing the energy per access for SPM and Instruction Cache, respectively.
- E_{mbst}, energy per burst for the main memory.
- E_{ovm}, energy consumed by instructions in overlay manager code.

OUTPUT:

- Set $\{S_1, S_2...S_r\}$, representing sizes of regions $R = \{R_1, R_2....R_r\}$, s.t. $\sum S_r \leq SPMSize$.
- Function-to-Region mapping, $X[f, r] = 1$, if f is mapped to region r, s.t. $\sum s_f \times X[f, r] \leq S_r$.

OBJECTIVE:

Minimize Energy Consumption for the given application. Given the GCCFG of an application, the objective is to create regions and function-to-region mapping such that when the application instrumented with this binary is executed on the given SPM, the total energy consumed is minimized. The total energy consumption is a summation of $E^{v_i}_{hit}$, (energy on SPM hit) and $E^{v_i}_{miss}$ (energy on SPM miss) where $v_i \in V_f$. While $E^{v_i}_{hit}$ consists of energy consumed by the overlay manager to check if the function v_i is present in SPM and energy consumed by the execution of the function from SPM, $E^{v_i}_{miss}$ has an additional energy component for moving the called function v_i from main memory to SPM and then moving the caller function back v_j on return. Code is transferred in burstsize of N_{mbst}. $nhit_{v_i}$ and $nmiss_{v_i}$ represents the number of hits and misses for the function v_i. The following equations characterizes the objective function

$$E^{v_i}_{hit} = nhit_{v_i} \times (E_{ovm} + E_{spm/access} \times s_i) \tag{1}$$

$$E^{v_i}_{miss} = nmiss_{v_i} \times (E_{ovm} + E_{spm/access} \times s_i + \frac{E_{mbst} \times (s_i + s_j)}{N_{mbst}}) \tag{2}$$

$$E_{total} = \sum_{v_i \in V_f} (E^{v_i}_{hit} + E^{v_i}_{miss}) \tag{3}$$

4 Our Approach

The goal of our approach is to use static analysis to dynamically map application code to regions on the SPM. Since the two sub-problems viz. region size determination and function-to-region mapping have a cyclic dependency, solving them independently will lead to sub-optimal results. Therefore, we require a technique to simultaneously solve the two sub-problems.

4.1 Overview

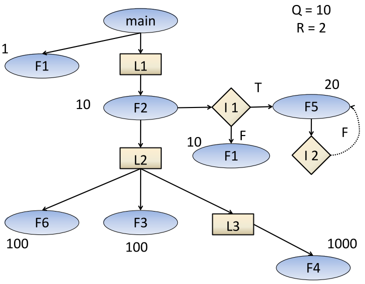
We first apply static code analysis to create a Global Call Control Flow Graph (GCCFG). Weights are assigned to nodes of the GCCFG , which is then transformed into an Interference Graph (I-Graph). The I-Graph and SPM size are then used as input to an ILP or SDRM heuristic to determine the number of regions and function-to-region mapping. The construction of GCCFG, weight assignment and I-Graph are explained in the following subsections with the help of an example shown in Figure 1(a).

```
MAIN ( )
    F1( )
    FOR
        F2 ( )
    END FOR
END MAIN

F5 (condition)
    IF ( condition )
        ......
    ELSE
        F5( condition)
    END IF
END F5
```

```
F2 ( )
    FOR
        F6 ( )
        F3 ( )
        WHILE
            F4 ( )
        END WHILE
    END FOR
    IF ( )
        F5 ( )
    ELSE
        F1( )
    END IF
END F2
```

(a) Example Code



(b) Global Call Control Flow Graph

Fig. 1. Construction of GCCFG

4.2 Construction of GCCFG

The GCCFG is an extension of the traditional Control Flow Graph (CFG) which is a representation of all paths that might be traversed through a function during its execution. A CFG is constructed for each function in the program and then all the CFGs are combined into a GCCFG in two passes. In the first pass the basic blocks are scanned for presence of loops (back edges in a dominator tree), conditional statements (fork and join points) and function calls (branch and link instructions). The basic blocks containing a loop header are labeled as L-node, those containing a fork point are labeled as I-node and the ones containing a function call are labeled as F-node.

If a function is called inside a loop, the corresponding F-node is joined to the loop header L-node with an edge. L-nodes representing nested loops, if any, are also joined. F-nodes not inside any loop are joined to the first node of the CFG. The first node, F-nodes , L-nodes and corresponding edges are retained, while all other nodes and edges are removed. Essentially this step trims the CFG, while retaining the control flow and call flow information. In this paper we assume that both paths, i.e. T and F edges, of a I-node will be executed, which is very similar to branch predication [7]. Therefore, although the GCCFG contain the I-nodes, the interference graph construction algorithm in Section 4.4 does not consider the presence of I-Nodes to determine the interference relationships between the F-nodes.

In the second pass, all CFGs are merged by combining each F-node with the first node of the corresponding CFG. Recursive functions are joined by a dashed edge. The merge ensures that strict ordering is maintained between the CFGs, i.e. if two functions are called one after another, the first function is a left child and the other function is a right child of the caller function. Thus the GCCFG is an approximate representation of the runtime execution flow of the program.

4.3 GCCFG Weight Assignment

For all F-nodes $v_f \in V_f$ of GCCFG, weights w_f, defaulting to unity, are assigned. The GCCFG in traversed in a top-down fashion. When an L-node is encountered, the weights of all descendent F-nodes are multiplied by a fixed quantum, Loop Factor Q. This ensures that a function which is called inside a deeply nested loop will receive a greater weight than other functions. For an F-node representing recursive function, the weight of the node is multiplied by a different fixed quantum, Recursive Factor R. This ensures that a recursive function will receive a greater weight than non-recursive ones. For the example shown in Figure 1(b), we choose Q = 10 and R = 2.

4.4 Interference Graph Construction

The weighted GCCFG has to be augmented considering the fact that if one function calls another function mapped to same region, then they will swap each other out during the function call and return back. Also if two functions mapped to same region are called one after another in the same nested level, then they will thrash excessively. Such functions are said to be interfering with one another and the GCCFG is not adequate to capture these interfering relationships. We transform the GCCFG into an

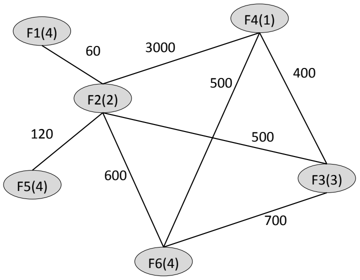
Algorithm 1. CONSTRUCT-IGRAPH (GCCFG = (V_f,V_l,E))

```
for $v_i$ = $v_1$ to ($v_f \bigcup v_l$) do
    for $v_j$ = $v_i$ to ($v_f \bigcup v_l$) do
        node = least-common-ancestor($v_i$,$v_j$)
        if (node == main) then
            relation($v_i$, $v_j$) = NULL ; cost [$v_i$,$v_j$] = 0;
        else if (node == L-Node) then
            relation($v_i$,$v_j$) = callee-callee-in-loop ; cost [$v_i$,$v_j$] = ($s_i$ + $s_j$) × MIN ($w_i$, $w_j$)
        else if (node == ($v_k \neq \{v_i, v_j\}$)) then
            relation($v_i$,$v_j$) = callee-callee-no-loop ; cost [$v_i$,$v_j$] = ($s_i$ + $s_j$) × MIN ($w_i$, $w_j$)
        else if (node == $v_i$ || node == $v_j$) then
            if (L-node in path from $v_i$ to $v_j$) then
                relation($v_i$, $v_j$) = caller-callee-in-loop ; cost [$v_i$,$v_j$] = ($s_i$ + $s_j$) × $w_j$
            else
                relation($v_i$, $v_j$) = caller-callee-no-loop ; cost [$v_i$,$v_j$] = ($s_i$ + $s_j$) × $w_j$
            end if
        end if
    end for
end for
```

I-Graph as outlined in Algorithm 1. Figure 2(a) shows the interference relationships and Figure 2(b) depicts the corresponding I-Graph between different nodes for the example GCCFG in Figure 1(b). In the next section we discuss an ILP and a heuristic which takes the *nodes* and the *cost* from the I-Graph as input and determines the region as well as the node (function)-to-region mapping.

NODE	NODE	INTERFERENCE RELATION
F2	F3	caller-callee-in-loop
F2	F4	caller-callee-in-loop
F2	F5	caller-callee-no-loop
F2	F6	caller-callee-in-loop
F3	F4	callee-callee-in-loop
F3	F6	callee-callee-in-loop
F4	F6	callee-callee-in-loop
F1	F2	caller-callee-in-loop

(a) Interference Relationships for the example GCCFG



(b) Interference Graph derived from the GCCFG

Fig. 2. Construction of I-Graph

5 Address Assignment: Where to Map

The problem of mapping functions-to-regions is a harder problem than the bin packing problem as the size of regions or bins is not fixed and each function (item to be placed in a bin) has an associated cost. Therefore, we propose a binary ILP and a heuristic to solve the 'where to map' problem.

5.1 Optimal Solution: Binary ILP

The input to the ILP is the I-Graph I = (V_f, E') constructed in previous section with s_i representing the size of node $v_i \in V_f$ and a $cost[v_i, v_j]$ associated with each edge (v_i, v_j). The output of the ILP is the function-to-region mapping $MAP : V_f \rightarrow R$, where R is the set of regions created. We define a binary integer variable $X[v_i, r]$ which is set to 1 if v_i is mapped to region r in SPM and set to 0 otherwise.

The cost of a region is the cost of placing two or more interfering nodes in the same region. The total cost is the summation of the cost of each region. The objective function to be minimized is the total cost of the interference graph which is given by (4) and subject to the constraints (5) and (6).

$$\text{Minimize} \quad \sum_{(v_i,v_j)\in E'} X[v_i, r] \times X[v_j, r] \times cost[v_i, v_j], \quad \forall r \in R \tag{4}$$

$$\sum_{r\in R} \max_{v_i \in V_f} (X[v_i, r] \times s_i) \leq SPMSize \tag{5}$$

$$\sum_{r\in R} X[v_i, r] = 1, \quad \forall v_i \in V_f \tag{6}$$

The first constraint (5) ensures that the sum of the sizes of all regions doesn't exceed the SPM size. The size of a region is the size of the largest function mapped to the region. Although the *max* function used above makes the constraint non-linear, it is linearized during implementation by making sure that all possible combinations of regions and functions mapped to the SPM does not exceed its size. The second constraint (6) ensures that a function is not mapped to more than one region. Because of the presence of two variables $X[v_i, r]$ and $X[v_j, r]$ in (4), the objective function is non-linear and cannot be modeled using LP. To make the above function linear, we introduce a new binary variable *U[v_i,v_j,r]* which is set to 1 if both v_i and v_j are mapped to same region r and set to 0 otherwise. The linearized objective function is given by equation (7).

$$U[v_i, v_j, r] \geq X[v_i, r] + X[v_j, r] - 1$$
$$U[v_i, v_j, r] \leq \frac{X[v_i, r] + X[v_j, r]}{2}$$

$$\text{Minimize} \quad \sum_{(v_i,v_j)\in E'} U[v_i, v_j, r] \times cost[v_i, v_j], \quad \forall r \in R \tag{7}$$

Since solving ILP may require prohibitively large computation resources, in the next section we propose a heuristic to solve the 'where to map' problem.

5.2 SDRM Heuristic

Our heuristic is based on the following observation. If two functions are joined by an edge in the I-Graph, then mapping them to the same region will incur a cost equal to the edge weight. The total cost of a region is the summation of edge weights

Algorithm 2. SDRM Heuristic

```
Overlay-I-Graph(I-Graph,SPM-Size)
global int num_regions = 0
global array address[]

 1. R[]: array of integer (size)
 2. node-address[]: array of integers
 3. sort-decreasing($E'$)
 4. for all e=($v_i$,$v_j$) in $E'$ do
 5.     for $v_k = v_i, v_j; v_k \leq$ SPM-Size do
 6.         if (node-address[$v_k$]==NULL) then
 7.             r = Determine-Region($v_k$)
 8.             node-address[$v_k$] = address[r]
 9.             R[r] = max(R[r], size($v_k$))
10.         end if
11.     end for
12. end for
13. return R and node-address
```

```
Determine-Region(Function $v_k$)
global int size_remaining = SPM-Size

 1. for all r in R, starting with least cost do
 2.     find r, s.t. e = ($v_k$,$v_j$) $\notin E'$, $v_j$ = MAP(r)
 3.     if ( found r) then
 4.         return r
 5.     end if
 6. end for
 7. if (size($v_k$) $\leq$ size_remaining) then
 8.     r = ++num_regions
 9.     address[r] = SPM-Size - size_remaining
10.     size_remaining - = size($v_k$)
11. else
12.     find r, s.t. cost of placing $v_k$ to r is min
13. end if
14. return r
```

of all such interfering functions. Algorithm 2 outlines the mapping procedure. The routine *Overlay-I-Graph* maps nodes of the I-Graph for the given size of the SPM. The output is the array *R* representing region sizes and array *node-address* representing the function-to-region mapping.. Line (3) sorts the edges of I-Graph in decreasing order of their weights. This ensures that the most interfering nodes are placed in separate non-overlapping regions of SPM if not constrained by the SPM size. It then calls the routine *Determine-Region* to find the region mapping for all unmapped nodes (4–7) and updates the corresponding region size after the node is mapped (8–9).

The routine *Determine-Region* determines the region for each unmapped node. It first checks if the node can be mapped to an existing region such that there is no interference with already mapped nodes in that region (1–6). If not, it checks if the node can be assigned to the remaining space, thereby creating a new region (7–10). Otherwise it finds an existing region such that the cost of the region after overlaying the node is minimum (12). In the worst case, all nodes will interfere with one another, complexity $O(E')$. Moreover the computation of the cost function will involve checking every node, complexity O(V_f). Hence the runtime complexity of the algorithm is O($V_f \times E'$).

6 Scratchpad Overlay Manager

The final step in the mapping process involves instrumenting the code with the mapping information obtained from SDRM or ILP and linking it with the SPM overlay manager (SOVM). The SOVM is responsible for keeping a track of function call and return during program execution. It has two data structures, the overlay table and region table. The overlay is filled with the mapping information during linking phase. The region

Table 1. Energy Model

Size(KB)	SPM(nJ)	4-way Cache(nJ)	Size(KB)	SPM(nJ)	4-way Cache(nJ)
0.5	0.107	0.534	4	0.145	0.551
1	0.128	0.538	8	0.173	0.564
2	0.134	0.542	16	0.206	0.587

table is used to keep a track of all functions currently residing in each region of SPM. Each function call and return statement in the application code is replaced by a stub function call to the SOVM. If the called function is not currently residing in SPM, the SOVM issues a direct memory access (DMA) command to transfer the function from main memory. The SOVM manager code then transfers the program control to the first instruction in the overlayed function. The SOVM and its data are mapped to the main memory to reduce the mapping pressure on the heuristic. Since the SOVM instructions and its associated data structures are fetched from the cache, we might see some runtime performance degradation. Our experiments show that the degradation is minimal.

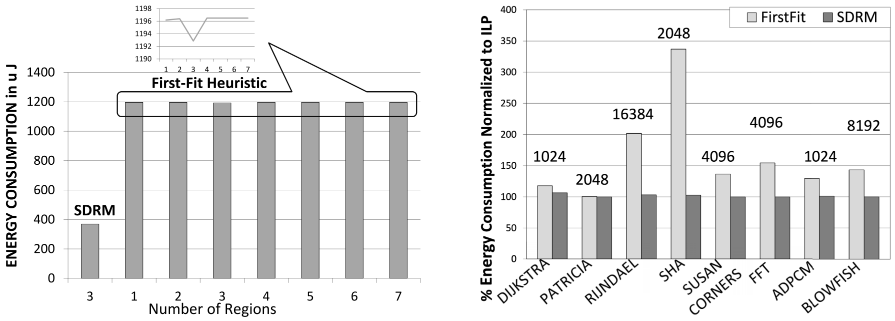
7 Experimental Setup

The instrumented binary is executed on a *cycle accurate simulator* that models an Intel XScale processor. The simulator has been augmented to model an on-chip SPM at the same level as instruction cache. The system modeled has a 32 KB instruction cache, 32 KB data cache, and a SPM, the size of which can be selected by the system designer. The simulator models a low power 32MB SDRAM from Micron as the main memory. The SPM is physically addressed and incoherent with the main memory subsystem. We perform our experiments on a set of embedded applications from [6]. The applications used and their respective code sizes are dijkstra(1588), patricia(2904), rijndael(21050), sha(2376), susan(46808), fft(4688), adpcm(1436), blowfish(9308). The per access energy numbers for SPM and I-Cache are given in table 1.

8 Results

8.1 First-Fit vs SDRM

In this section we present a comparison of total energy consumption between the ILP, SDRM and the first-fit heuristic for various benchmarks. For first-fit we assume that the SPM is divided into variable sized regions (Experimentally we found that variable-sized region gives better results than equal-sized region). The previous approach does not precisely state a way of finding these region sizes. To be unbiased, we performed an exploration for various sizes and number of regions. For example, for x bytes of SPM, we divided it into x/2, x/4, x/8,... x/r, where the value of r was found by exploration. The regions were considered in the same order for allocation. Figure 3(a) show the first-fit energy consumption trend for *sha* as we explore the number of regions for a 2KB SPM. As shown in the graph, there is an optimal number of regions $(r = 3)$ in first-fit, at



(a) SHA Benchmark: first-fit heuristic with varying number of **variable** sized regions

(b) Energy Comparisons between ILP, SDRM and first-fit for various benchmarks

Fig. 3. First-Fit vs SDRM

which the energy consumption is minimum. For smaller number of regions $(r < 3)$, not all interfering functions can be mapped, since the number of such functions is higher than the number of regions. Some functions are spilled to main memory, resulting in a higher energy consumption. As we increase the number of regions, more functions will be overlayed and the energy consumption decreases, reaching a local minimum at $(r = 3)$. However, if we compare this value with the first bar which indicates the energy consumption from SDRM mapping, it is significantly higher. The reason is that the critical function for sha does not fit into any region of the SPM, corroborating our argument that pre-determining the number of regions does not lead to optimal solution. Further increase in number of regions $(r > 3)$ fragments the SPM into smaller sized regions. As large sized functions cannot fit, this again results in spilling of such functions to the main memory which causes a rise in energy consumption. On further increase $(r > 4)$, the SPM gets more fragmented, but the mapping does not change and there is no change in energy consumption.

To the best of our knowledge, none of the previous approaches have demonstrated any technique for finding the optimal number of regions at which the energy consumption would be minimal. The only way to find this number is by exploration of the entire solution space by varying the number and size of regions. The search space can be reduced by smart exploration techniques, but only up to a limited extent as the exploration process is a time consuming task involving recompilation and execution of program every time. The SDRM technique proposed does not incur this expense as it simultaneously finds the optimal number of regions and their sizes while solving the mapping problem. The first bar in the graph shows the energy results obtained by SDRM for a 2KB SPM. SDRM divides the SPM into three variable sized regions and exhibit a 69% energy reduction compared to first-fit which divides the SPM into three variable sized but pre-determined regions.

Figure 3(b) shows the comparison of energy consumption between SDRM and first-fit heuristic for various benchmarks, normalized to the ILP energy values. The optimal number and size of the regions for first-fit are found by exploration as discussed previously. From the figure, we observe that the energy for SDRM is always close to 100%, indicating that the solution obtained from the SDRM heuristic is close to the optimal ILP solution. Moreover, the maximum energy reduction is observed for *sha*, where the first-fit performs poorly, as the most critical region does not even fit into any region. On the other hand, since the SDRM does not predetermine the region sizes, the critical functions are always mapped to some region of the SPM. On an average we observe a 27% energy reduction for SDRM compared to the first-fit technique.

8.2 Cache-Only vs. Horizontally Split Architecture

In this experiment, we compare our mapping technique for a HSA against a COA. The COA architecture consists of 2x bytes of I-cache while the HSA consists of x bytes of SPM and x bytes of I-cache. Figure 4 shows how the HSA architecture with SDRM technique performs in comparison with a COA for *sha* benchmark.

For small sizes of SPM, the critical functions do not fit into the SPM at all and are spilled to cache. Hence there is no significant difference between a COA and a HSA. As we increase the size to 2048 bytes, all functions can fit into the SPM, and the functions would need to be overlayed as the aggregate size of 2376 bytes for *sha* is greater than 2048 bytes. At this size of SPM, we see a significant reduction in energy as all the code is fetched and executed from the SPM instead of the I-cache. At a larger size of 4096 bytes, all the functions can be mapped onto the SPM without any overlay, which means no calls to the SPM overlay manager and no runtime overhead due to DMA transfer instructions. We should therefore have observed a further decrease in energy consumption. However, since we assume a model in table 1 where the energy per access for SPM increases with size, we observe an increase in energy consumption with increasing size of SPM. For *sha* benchmark the HSA shows a reduction of 77% compared to the COA. The average reduction is 35% across all benchmarks.

This experiment demonstrates the effectiveness of a split memory subsystem architecture when supported by an intelligent mapping technique like SDRM. In other words, given an architecture with only an instruction cache, we can always reduce the energy consumption by splitting the power hungry instruction cache equally into a SPM and a smaller instruction cache. A pure compiler technique like SDRM can then be used, requiring just a simple recompilation of the application, with no profiling overhead.

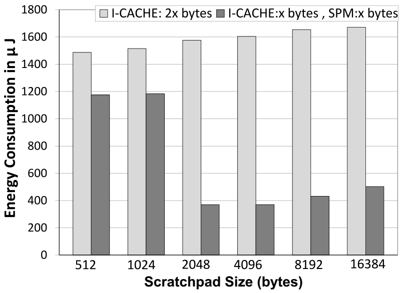


Fig. 4. SHA: Energy comparisons between COA and HSA SDRM

8.3 Performance Overhead

Since the SOVM code is fetched and executed from the I-cache, there is a performance penalty in terms of runtime cycles due to the extra instructions. One way to reduce this overhead would be to map the SOVM code to the SPM instead of cache. However, this would mean less space available to map the functions themselves resulting in a potential spill of some critical functions to the cache, which means a greater energy consumption. An additional penalty is incurred due to clearing of the branch target buffer table, each time an overlayed function is transferred from main memory to SPM. This is essential, otherwise branch instructions would jump to invalid addresses from the previous overlayed function, thereby crashing the application. There is also an additional penalty due to stalls during code transfer from main memory to SPM. We observe an average performance degradation of 1.9% across all benchmarks.

9 Conclusion

In this paper, we presented a fully-automated, dynamic, code overlaying technique based on pure compiler analysis for energy reduction for on-chip scratchpad memories in embedded processors. We formulated an ILP which gives an optimal solution and a heuristic which gives a near-optimal solution and simultaneously addresses both the important issues of region size determination and function-to-region mapping. The proposed technique and HSA architecture succeeds in achieving a greater energy reduction against a previous approach and a unified instruction COA architecture, respectively. Compared to the best performing previously known heuristic our approach achieves an average energy reduction of 27% with an average performance degradation of just 1.9%. We also demonstrated that by splitting the I-cache into equal sized smaller I-cache and SPM and using a pure compiler technique like SDRM, we can always reduce the total energy consumption. This paves the path of reducing the memory subsystem energy even in general purpose processors employing the split architecture.

References

1. Angiolini, F., Menichelli, F., Ferrero, A., Benini, L., Olivieri, M.: A post-compiler approach to scratchpad mapping of code. In: Proc. CASES, pp. 259–267 (2004)
2. Avissar, O., Barua, R., Stewart, D.: An optimal memory allocation scheme for scratch-pad-based embedded systems. Trans. on Embedded Computing Sys. 1(1), 6–26 (2002)
3. Banakar, R., Steinke, S., et al.: Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In: Proc. CODES, pp. 73–78 (2002)
4. Egger, B., Lee, J., Shin, H.: Scratchpad memory management for portable systems with a memory management unit. In: Proc. EMSOFT, pp. 321–330 (2006)
5. Montanaro, J., Witek, R.T., et al.: A 160-mhz, 32-b, 0.5-w cmos risc microprocessor. Digital Tech. J. 9(1), 49–62 (1997)
6. Guthaus, M., Ringenberg, J., et al.: Mibench: A free, commercially representative embedded benchmark suite. In: Workshop on Workload Characterization, pp. 3–14, December 2 (2001)
7. Smith, J.E.: A study of branch prediction strategies. In: Proc. ISCA, pp. 135–148 (1981)

8. Steinke, S., Wehmeyer, L., Lee, B., Marwedel, P.: Assigning program and data objects to scratchpad for energy reduction. In: Proc. DATE, p. 409 (2002)
9. Steinke, S., Grunwald, N., et al.: Reducing energy consumption by dynamic copying of instructions onto onchip memory. In: Proc. ISSS, pp. 213–218 (2002)
10. Udayakumaran, S., Dominguez, A., et al.: Dynamic allocation for scratch-pad memory using compile-time decisions. Trans. on Embedded Computing Sys. 5(2), 472–511 (2006)
11. Verma, M., Marwedel, P.: Overlay techniques for scratchpad memories in low power embedded processors. IEEE Trans. on VLSI 14(8), 802–815 (2006)

An Utilization Driven Framework for Energy Efficient Caches

Subramanian Ramaswamy and Sudhakar Yalamanchili

Center for Experimental Research on Computer Systems
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
ramaswamy@gatech.edu, sudha@ece.gatech.edu

Abstract. With the shift from scaling frequency to scaling the number of cores, efficiency becomes a primary design consideration. The ability to scale the number of cores while pushing back the memory and power walls with small increases in die size will require significant improvements in cache efficiencies. This paper provides strategies to improve L2/L3 data cache efficiencies by coupling voltage scaling with flexible cache management policies. Specifically, we propose a framework that encompasses i) off-line creation of a voltage-area profile, ii) on-line measurement of cache line utilization to drive voltage scaling, and, iii) changing the placement function to match the voltage-scaled area and the program-phase cache footprint. The proposed techniques were applied to several benchmarks resulting in performance efficiencies doubling, energy efficiencies improving by 10% (relatively) with a 10% improvement in Energy Delay Product.

1 Introduction

The shift from scaling frequency to scaling the number of cores continues stressing the off-chip memory bandwidth and demands larger on-chip caches. Although large caches push back the memory wall, they are inefficient consumers of energy. For a range of application domains, data cache utilizations have been found to be below 20%, performance efficiencies below 10% and energy efficiencies in the range 1-5%! [1]. These low efficiencies are not sustainable for a critical architectural resource that is a dominant on-chip resource consumer, consuming over 70% of the transistor budget [2] and 15-30% of the energy budget [3,4]. Future growth in execution performance will need more cores while the increased memory bandwidth required to sustain these cores necessitates larger caches; both requirements must be met with small increases in die sizes. Consequently, the ability to scale the number of cores while pushing back the memory and power walls requires significantly improved cache efficiencies.

At deep sub-micron geometries, the energy consumption in L2/L3 caches is dominated by leakage [4,5,6]. Our approach to improving energy efficiency primarily relies on dynamically *sizing* the cache to match the application memory footprint or working set during a program phase; all remaining cache lines are powered down. Our approach to sizing differs from past approaches in that, our techniques produce fully functional smaller caches by concurrently changing the placement function - the manner in which

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 583–594, 2008.

main memory lines share cache resources. We refer to this as *shaping* the cache. Therefore, there are no references to inactive cache lines in a resized cache.

Sizing and shaping are used together to improve L2/L3 cache efficiency. The key insight is to combine sizing and shaping with voltage scaling in the cache. As voltages are scaled down, the defect-free failure rate for cache memory cells increases, (e.g., due to timing failures [7]). Additionally, program phases have varying cache footprints. These insights are combined into an off-line generated voltage-sizing profile of the cache and this profile is traversed dynamically using on-line utilization measurements. The utilization measurements are used to *up-size/down-size* the cache, i.e., pick a new point on the voltage-size profile. Each change in the cache size is accompanied by a refinement of the placement function to map memory to the active cache lines. Companion measurements of miss rates are used to modulate sizing decisions to prevent energy savings that carry performance penalties, i.e., significantly higher miss rates.

The next section reviews an efficiency model proposed earlier and used to evaluate the implementation proposed here, as well as some insights into behaviors that affect efficiency (as opposed to raw performance). Section 3 describes the operational model while Section 4 details the techniques for i) on-line utilization measurement, ii) sizing, and iii) cache shaping. The paper concludes by evaluating the implementation and providing directions for future work.

2 Modeling and Measuring Efficiency Metrics

To keep the paper self contained, the efficiency model and metrics originally provided in [1], are reproduced here.

2.1 Definitions and Relationships

At a clock cycle, a cache line may be *active* (powered) or *inactive* (powered down). A cache line is *live* at a clock cycle if it contains data that will be used prior to eviction, and it is *dead* otherwise [8,9]. Thus, on any clock cycle, a cache line is live, dead, or inactive. For a cache with L lines over T cycles, the total cache cycles expended is the sum of the *live cycles*, the *dead cycles*, and the *inactive cycles*.

Cache utilization, η_u, is the average percentage of cache lines containing live data at a clock cycle [8,9]. Utilization is computed as shown in Equation 1.

$$\eta_u = \frac{\sum_{i=0}^{i=L-1} live\_cycles_{line_i}}{\sum_{i=0}^{i=L-1} active\_cycles_{line_i}} \tag{1}$$

The *effectiveness* of the cache, E, is the percentage of cache cycles devoted to live lines and is shown in Equation 2. Equation 2 can also be written as Equation 3. Effectiveness serves as a metric for comparing programmed cache line shutdown strategies; the higher the effectiveness, the higher the percentage of the active cache that retains live data.

$$E = \frac{\sum_{i=0}^{i=L-1}(live\_cycles_{line_i} + inactive\_cycles_{line_i})}{\sum_{i=0}^{i=L-1}(active\_cycles_{line_i} + inactive\_cycles_{line_i})} \tag{2}$$

$$E = 1.0 - \frac{\sum_{i=0}^{i=L-1} dead\_cycles_{line_i}}{total\_cycles * L} \tag{3}$$

The most effective scheme is one where all cache cycles are either live or inactive. Effectiveness is equivalent to utilization without any energy management. An efficient cache must be *effective* with high performance. Cache *performance efficiency,* η_p, is defined in Equation 4 as the product of effectiveness and a scaling factor, where t_c is the cache access time, t_p is the miss penalty, and m is the miss rate. A cache has $\eta_p = 1$ if it does not contribute any dead cycles and has a 100% hit rate.

$$\eta_p = E * \frac{t_c}{t_c + m * t_p} \tag{4}$$

Energy efficiency, η_e, is the ratio of *useful work* to total work. Useful work is the switching energy expended in a cache hit. The total work is the sum of the switching energy consumed during all cache accesses (hits and misses) and the leakage energy. A cache has $\eta_e = 1$ if all the energy consumed by the cache is equal to the switching energy consumed during cache hits. Energy efficiency is defined in Equation 5, where sw_{energy} represents the switching energy and $leak_{energy}$ represents the leakage energy. The switching energy for cache hits and misses are assumed to be equal; this assumption affects the results negligibly.

$$\eta_e = \frac{sw_{energy} * num_{hits}}{sw_{energy} * (num_{hits} + num_{misses}) + leak_{energy}} \tag{5}$$

Although cache sets or lines may be powered down to reduce leakage energy, additional misses that may result from the powering down of parts of the cache can increase program execution times and thus higher overall energy consumption.

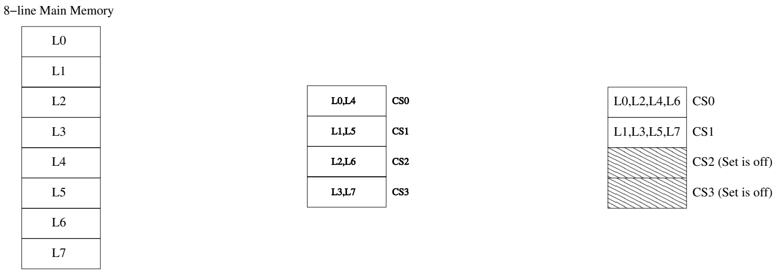
2.2 Improving Efficiency

Results that motivate the proposed techniques are low utilizations ($< 20\%$), performance ($< 10\%$) and energy efficiencies (1-5%). This indicates that the majority of the cache energy and area costs are spent in maintaining *dead* lines, and that the re-use of live lines was low, and the ratio of leakage energy to switching energy was very high.

To increase efficiency we have to i) reduce the number of dead lines in the cache, and, ii) make better use of active lines. The first step improves energy efficiency and the second improves performance efficiency. The approach reported here differs from our prior efforts in i) coupling voltage scaling with resizing and remapping, and ii) creation of a static voltage-sizing profile that is dynamically traversed, with iii) cache shaping performed dynamically using a model employing utilization and miss-rate. Thus, caches are one-time reconfigured to produce this profile (post manufacturing) and this profile is traversed in an application-specific manner. Specifically we identify triggers to move to a new voltage-sizing point concurrently with the computation of a new placement function.

3 Operational Model

A *conflict set* is the set of main memory lines that is mapped to a cache set. They are constructed using *modulo* placement for modern caches as illustrated in Figure 1(b),



(a) Eight line main memory

(b) Modulo placement

(c) Energy Efficient Cache

Fig. 1. Conflict set construction

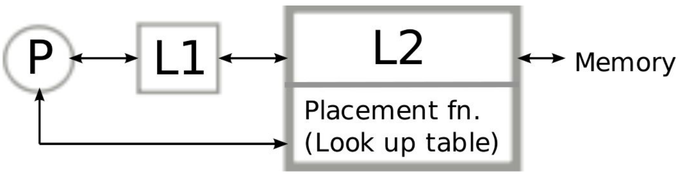


Fig. 2. Architecture Model

where a memory line at address L is placed in the cache set L mod S, with S sets in the cache. Modulo placement exploits spatial locality and maps contiguous memory locations (lines) to distinct cache sets. Shaping customizes this placement function as shown in Figure 1(c).

Customized placement is implemented for the L2 cache as shown in Figure 2. The L2 access is remapped through a lookup table to implement customized placement. Operationally, when a down-sizing or up-sizing operation is performed, the remapping table is reloaded with address translations, which is performed in software. We also assume the existence of the ability to turn off cache lines using the *Gated-Vdd* approach proposed by Powell *et al.*[10] which enables turning off the supply voltages to caches lines, and has an additional area cost of 3%. The area and energy costs of the lookup table is addressed in Section 5 and is negligible.

4 On-Line Cache Management

Improving performance and energy efficiencies relies on the following: off-line characterization, on-line computation of cache utilization, cache sizing, and, cache shaping.

4.1 Off-Line Characterization

Several studies have documented the challenges of the manufacturing process to fabricate devices with design tolerances because of the significant variations in transistor device characteristics within a die (WID), across dies (D2D), and between wafers

(W2W) [11,12]. We assume i) the existence of built-in-self-test (BIST) capability for the cache as described in Agarwal *et al.*[13], ii) operation of the L2 cache as a separate voltage island (for example, the Barcelona die has separate voltage island for the L3 cache), and iii) the availability of four voltage levels. The BIST is operated off-line at each voltage level to identify fault free sets. A cache set is marked as failed if one cache line within it is marked failed - this is a line with at least one failed bit cell. Failures are assumed to be monotonic, i.e., a line marked faulty at a voltage level cannot be operational at a lower voltage level. This information is captured in a voltage-sizing map that reflects available fault free cache sets at each voltage level. This map can be made more aggressive by further down-sizing the cache at each voltage. The following sections deal with *when* to resize (equivalently change voltage levels and power down unused lines) and *how* to remap memory to the active cache sets.

Due to the lack of real failure rate data, we use two synthetic voltage-area maps. The first provides caches sized at 25%, 50%, 75%, and 100% of the original cache at respective voltage levels (i.e., the highest level is associated with a fully sized cache). The second map utilizes cache sizes of 60%, 70%, 80%, and 100%. This map is informed in part by our prior work [14] that noted significant performance penalties when the unusable cache size exceeded 40%. Utilization values uniformly index both maps, i.e., a measured average line utilization of 50% would index into the either a cache size of 50% or 70% respectively.

4.2 Online Computation of Cache Line Utilization

We found that effective sampling of the activity in the data cache provides a reasonable estimate of utilization. Within each set, the last hit is recorded via a hit status bit associated with each line. The status bit will be equal to 1 for the last line hit in the set, and 0 for all other lines in the set. The hit status bits of all lines are sampled every S cycles. The utilization is computed every $W > k \times S$ cycles for some integer k. The utilization of a line is determined by the number of times its hit status bit is set in this interval W. If it is k, the utilization for that line is 100%. This cache line utilization averaged across all cache lines is the measured global cache utilization. Our simulations used a value of one million cycles for one sampling period, and used 5 million L2 cache references as the value for W. The sampled values are within 10% of the actual utilization values in absolute terms for all benchmarks (for the majority of the benchmarks the measured utilization is within 5% of actual). Future work will focus on automatic phase detection schemes such as those described in [15]. Since we sample the cache infrequently, our sampling and the computation of utilization is done through software. The software cost of the sampling and computation ($< 1\%$) is negligible compared to the execution time and energy savings.

4.3 Cache Sizing

The measured utilization at the end of each phase (5M references) is used to select a cache size/voltage by indexing the voltage-sizing map. Due to the lack of real failure rate data, we use two synthetic voltage-area maps as explained in Section 4.1. Thus, at the end of a phase, the average measured cache utilization is used to index a map to obtain a new cache size, and a new placement function is computed and loaded into the address remapping table in the L2.

4.4 Shaping the Cache

Shaping the cache is comprised of two steps - i) computing the placement function for a specific cache size, and ii) configuring the cache to implement this new mapping.

Computing the Placement Function. At the end of a phase, the global cache utilization across all sets is computed. This utilization is used to index the voltage-sizing map which identifies the number of sets that will be active in the next phase. The conflict sets of the cache sets to be powered down must be merged with conflict sets that map to cache sets that are active in the next phase. Merging takes place between active cache sets that had the lowest utilization in the last phase. In our current implementation only one cache up-sizing operation is performed - to full cache operation. The reason for this constraint was driven primarily by empirical evidence that up-sizing tended to occur on phase boundaries and fine grained up-sizing was not of much benefit for these benchmarks.

For comparison, we also evaluate nearest neighbor merging described in Agarwal *et al.* [13] where only the pairs of conflict sets that map to adjacent sets in the cache are merged. This approach simpler in that no status information is maintained nor is any online decision made and is Option 2 in Figure 3. However, both techniques are invoked using the same criteria.

Configuring the Cache. The remapping is implemented by updating the address remapping table which is illustrated in Figure 4. The configuration update is performed in software. The hardware logic requires BIST data that contains information on which sets fail at what voltage level. A set of four mask registers contain the status of each cache set at each of the four voltage levels. For a given cache set, the voltage level indicator indexes the mask bits for the set. The selected mask bit is used to power down the corresponding set or leave it powered on. For a 256KB cache with 256 sets, these mask registers will use, in total, $256 * 4$ bits. Additionally, hit status bits have to be maintained per cache line (2048 bits for a 256KB 8-way 128 byte L2 cache).

The major investment in area comes from the lookup table used for remapping. For the 256 KB L2 used in our experiments, 256, byte-wide entries are required for this look up table. The new tag is the old tag augmented with the original cache set index because the cache tag array may contain tags from customized as well as modulo placement. This adds another 256*8 bits, resulting in a total additional storage of 896 bytes for customized placement, which is insignificant compared to the L2 cache size. The energy consumption of this additional logic is less than 1% of the total cache energy consumption. When the placement is changed, cache lines that had multiple conflict sets mapping to it are written back if dirty and invalidated to avoid any synonym or aliasing problems. The address remapping is assumed to be accessible to the compiler via special custom instructions.

In terms of latency, an additional cycle is required for the address translation logic. While this can be easily masked by performing the lookup in parallel with an access to

```
Input: $util[\,], utilization, missrate$
Output: $remap[\,]$
$initialize(trigger\_old, trigger\_new, new\_voltageindex, old\_voltageindex)$
$trigger\_new = utilization * t_c/(t_c + t_p * missrate)$ { This is the metric used to detect phases in memory behavior;this combines utilization and miss rate}
if $trigger\_new > trigger\_old$ then
    $new\_voltageindex = int(utilization)/25$ {Defining four voltage indices based on utilization, with index 3 corresponding to the highest voltage level}
else
    $turn\_on\_all\_sets(); new\_voltageindex = 3$ {if performance is suffering more than the savings in utilization, turn all sets back on}
end if
if $new\_voltageindex > old\_voltageindex$ then
    $new\_voltageindex = old\_voltageindex$ {We do not up-size the cache in steps but turn all sets back on, whereas down-sizing can occur in steps}
end if
$turn\_off(new\_voltageindex)$ {schedule cache sets to turn off based on corresponding voltage level which were identified during BIST}
for all faulty cache sets , $fc$ , at $new\_voltageindex$ do
    OPTION 1: $remap(fc) = faultfree\_nextneighbor\_set$
    OPTION 2: $remap(fc) = faultfree\_lowestutil\_set$
    OPTION 2: $update(util[\,])$ { the utilization array is updated with the utilization of the set being remapped being added to the set to which it is remapped}
end for{ Option 1 signifies the next neighbor strategy, whereas Option 2 is the utilization driven remapping strategy}
$trigger\_old = trigger\_new$
$old\_voltageindex = new\_voltageindex$
```

Fig. 3. Shaping Algorithm using Utilization

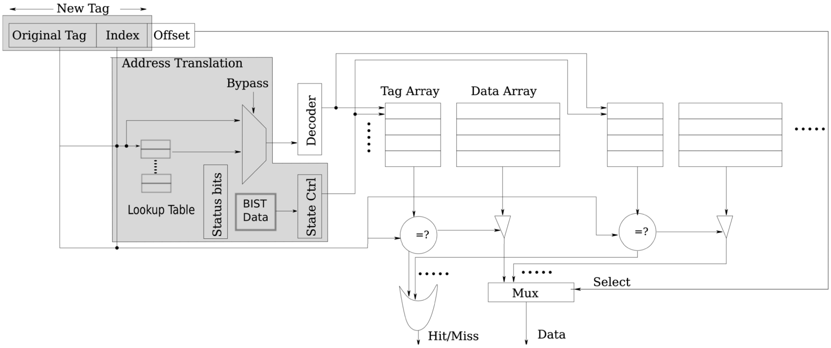


Fig. 4. Hardware Implementation

the L1 cache, even if the lookup was not performed in parallel with the L1 cache access, the additional cycle does not alter execution time by more than 2% in our simulations.

5 Performance Evaluation

5.1 Simulation Methodology and Assumption

We simulated the execution of a subset of applications from the SPEC2000 suite [16], DIS suite [17] and Olden [18] using the $Simplescalar$ 3.0 [19] simulator.

We obtained energy estimates using $Cacti$ 5.1 [20] for 70 nm technology (with the latest version using the ITRS-HP technology models). We assume the L2 cache access latency to be fixed at 15 cycles. Our definition assumes that the read and write energies are the same—which increases energy efficiency compared to a more precise definition. Leakage energy is predominant and cache writes constitute a small fraction of the total number of accesses, (for example, Tarjan *et al.*[21] estimates that at 70 nm, greater than 95% of the total cache power is leakage) therefore this assumption affected efficiency by less than 1%, as given by $Cacti$ estimates. The energy was calculated with the cache operating at the highest expected frequency as given by $Cacti$ estimates. The L1 cache configuration was 16KB 2-way with 64-byte lines. The results are for the cache only, and do not include impact of voltage scaling on the processor datapath.

5.2 Results and Discussion

The five schemes represented in the figure are as follows:

1. No energy management representing traditional caches
2. Next neighbor scheme (1) representing the next neighbor scheme with cache sizing maps of 60, 80, 90 and 100%
3. Customized Shaping scheme (1) representing the utilization based remapping scheme with cache sizing maps of 60, 80, 90 and 100%
4. Next neighbor scheme (2) representing the next neighbor scheme with cache sizing maps of 25, 50, 75 and 100%
5. Customized Shaping scheme (2) representing the utilization based remapping scheme with cache sizing maps of 25, 50, 75 and 100%

From Figure 5, it can be observed that effectiveness has increased significantly across all applications with all management schemes. The next neighbor scheme perform as well as the customized shaping schemes in terms of effectiveness – this can be explained by the utilization tracking mechanism which predicts shut down phases well.

However, the performance of the schemes diverge when performance efficiencies are compared, as shown in Figure 6. When performance efficiencies are compared, the customized shaping schemes relatively outperforms the next neighbor scheme on average by 10% (Note that a 1% difference in performance efficiency represents a 10% improvement). Compared to the traditional caches, performance efficiencies are more than doubled in many benchmarks.

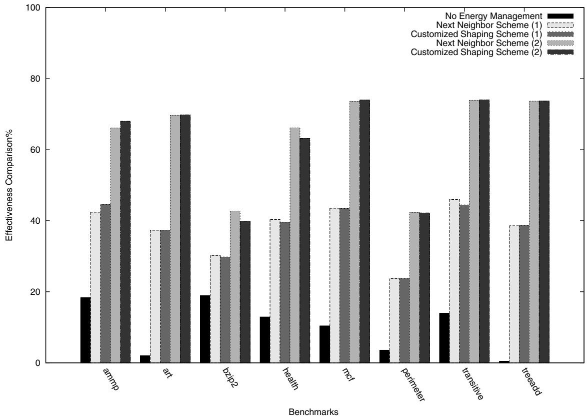


Fig. 5. Effectiveness comparison

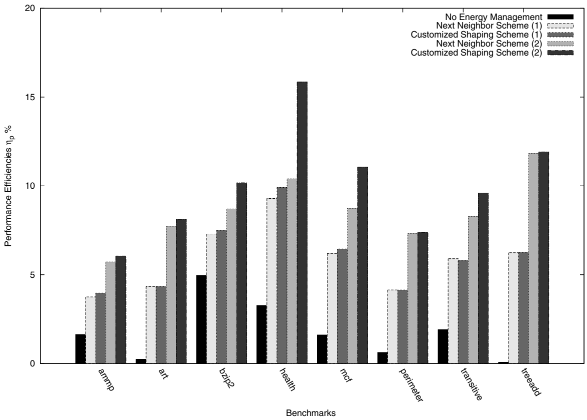


Fig. 6. Performance Efficiency comparison

The comparison of energy efficiencies are shown in Figure 7. Again, it is seen that energy efficiency increases are broad based over the traditional cache for many benchmarks. Though the energy efficiency improvements as an absolute percentage seem modest, it has to be factored in that there are some fundamental constraints while attempting to improve energy efficiency. Given, that an L2 cache line is accessed only once every few million cycles, leakage energy dominates. A 1% absolute improvement in energy efficiency is a 25% increase — this has to be factored in the evaluation. The customized shaping scheme consistently outperforms the next neighbor schemes by 0.05–1.0%, representing an average absolute improvement of 0.5% (again, this is a 10-15% improvement relative to the values of energy efficiency).

The increase in energy efficiency is captured by looking at the Energy Delay Product as shown in Figure 8. EDP is calculated for the cache alone, since it is assumed to be in a separate voltage domain. EDP for most benchmarks are reduced by more than

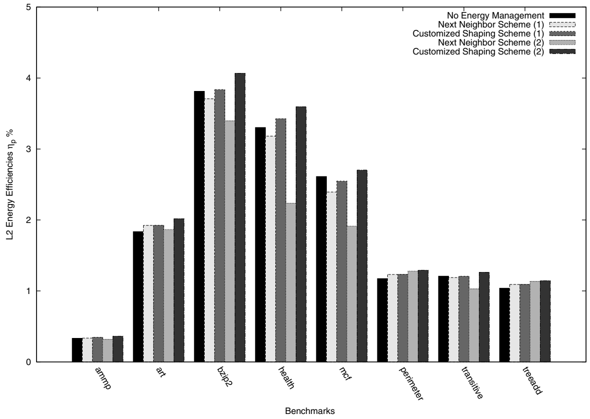


Fig. 7. Energy Efficiency comparison

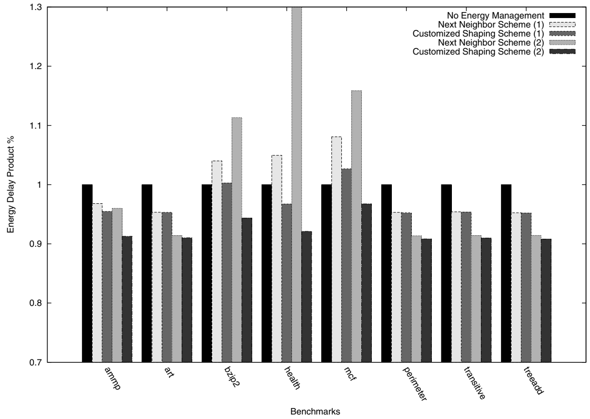


Fig. 8. Energy Delay Product comparison

15% using the customized shaping scheme with *no EDP regressions* for any benchmark. The next neighbor approach, however suffers significant regression for many benchmarks;for example, the EDP for benchmark *health* doubled using next neighbor (this is not represented in the figure). In summary, both energy management schemes perform well using the online utilization tracking mechanism, whereas the customized shaping scheme is able to provide an additional 10-15% relative improvements in energy efficiencies and EDP.

6 Related Work

The various techniques for reducing leakage in instruction and data caches include turning off cache lines, sets or ways (examples include Abella *et al.*[22], Kaxiras *et al.*[9], Powell *et al.*[10], Zhang. C *et al.*[23], Zhang M. and Asanovic [4] and Zhou *et al.*[24]) or putting cache lines in a drowsy state as proposed by Flautner *et al.* [5], etc. Zhang W. *et al.*[3,25] use special instructions to schedule instruction cache turn offs using loop and branch information or maintain the cache in a drowsy state activating cache lines prior to access. Whenever cache sets are turned off, generally an access to the cache set results in an expensive memory access in addition to the power-up latency.

In contrast, our approach first sizes fully functional caches that avoids costly accesses to inactive cache lines, although we have to manage the miss rate effectively to avoid substantial performance impact. This is achieved via shaping customized to the application reference behavior. Our heuristics are based on merging conflict sets which permits our techniques to be more aggressive in scheduling power-down events.

Relative to our prior work [1,14], the approach here is distinct in that it i) integrates voltage scaling, ii) computes a static profile of voltage-sizing behavior, and iii) proposes to dynamically traverse this profile using run-time measured utilization. In addition we propose a low cost, and effective approach for the run-time computation of cache set/line utilization that could readily be used to drive other cache management policies.

7 Conclusion

Scaling the number of cores for performance places a premium on cache sizes. Large L2/L3 caches are inefficient consumers of energy and area therefore buying performance with larger caches is no longer feasible. We believe that efficiency will be a major driver for future processor design. This paper presents a framework and implementation of an approach to make substantive improvements in the efficiency of on-chip caches. Specifically, we couple voltage scaling with more flexible cache management policy (placement) to realize gains in cache efficiency. We achieve about a 10% improvement in cache energy efficiency resulting in an EDP improvement of 10%.

References

1. Ramaswamy, S., Yalamanchili, S.: Improving cache efficiency via resizing + remapping. In: ICCD (2007)
2. Ranganathan, P., Adve, S., Jouppi, N.P.: Reconfigurable caches and their application to media processing. In: ISCA 2000, pp. 214–224 (2000)
3. Zhang, W., Hu, J.S., Degalahal, V., Kandemir, M., Vijaykrishnan, N., Irwin, M.J.: Compiler-directed instruction cache leakage optimization. In: MICRO 35, pp. 208–218 (2002)
4. Zhang, M., Asanovi, K.: Fine-grain CAM-tag cache resizing using miss tags. In: ISLPED, pp. 130–135 (2002)
5. Flautner, K., Kim, N., Martin, S., Blaauw, D., Mudge, T.: Drowsy caches: Simple techniques for reducing leakage power. In: ISCA (2002)
6. Zhang, W., Hu, J.S., Degalahal, V., Kandemir, M., Vijaykrishnan, N., Irwin, M.J.: Compiler-directed instruction cache leakage optimization. In: MICRO 35 (2002)
7. Mukhopadhyay, S., Mahmoodi-Meimand, H., Roy, K.: Modeling of failure probability and statistical design of SRAM array for yield enhancement in nanoscaled CMOS. IEEE Trans. on CAD of Integrated Circuits and Systems 24(12), 1859–1880 (2005)

8. Burger, D.C., Goodman, J.R., Kagi, A.: The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Technical Report UWMADISONCS CS-TR-95-1261, University of Wisconsin, Madison (January 1995)
9. Kaxiras, S., Hu, Z., Martonosi, M.: Cache decay: exploiting generational behavior to reduce cache leakage power. In: ISCA, pp. 240–251 (2001)
10. Powell, M., Yang, S.H., Falsafi, B., Roy, K., Vijaykumar, T.N.: Gated Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In: ISLPED, pp. 90–95 (2000)
11. Borkar, S., Karnik, T., Narendra, S., Tschanz, J., Keshavarzi, A., De, V.: Parameter variations and impact on circuits and microarchitecture. In: DAC, pp. 338–342 (2003)
12. Borkar, S.: Design challenges of technology scaling. IEEE Micro 19(4), 23–29 (1999)
13. Agarwal, A., Paul, B.C., Roy, K.: A novel fault tolerant cache to improve yield in nanometer technologies. In: IOLTS
14. Ramaswamy, S., Yalamanchili, S.: Customized placement for fault tolerant caches in embedded processors. In: ICCD (2006)
15. Sherwood, T., Perelman, E., Hamerly, G., Sair, S., Calder, B.: Discovering and exploiting program phases. IEEE Micro 23(6), 84–93 (2003)
16. The Standard Performance Evaluation Corporation: The SPEC CPU 2000 Benchmarks (2000)
17. Titan Systems Corporation: DIS Stressmark Suite - Specifications for the Stressmarks of the DIS Benchmark Project v 1.0. Technical report (2000)
18. Rogers, A., Carlisle, M.C., Reppy, J.H., Hendren, L.J.: Supporting dynamic data structures on distributed-memory machines. ACM TOPLAS 17(2), 233–263 (1995)
19. Burger, D., Austin, T.: The SimpleScalar tool set, version 3.0. Technical report, Computer Sciences Dept., University of Wisconsin-Madison (1999)
20. Thoziyoor, S., Muralimanohar, N., Jouppi, N.P.: Cacti 5.0. Technical report
21. Tarjan, D., Shyamkumar Thoziyoor, N.P.J.: Cacti 4.0. Technical report
22. Abella, J., Gonzalez, A., Vera, X., O'Boyle, M.F.P.: Iatac: a smart predictor to turn-off l2 cache lines. ACM Trans. Archit. Code Optim. 2(1), 55–77 (2005)
23. Zhang, C., Vahid, F., Najjar, W.: A highly configurable cache architecture for embedded systems. In: ISCA, pp. 136–146 (2003)
24. Zhou, H., Toburen, M.C., Rotenberg, E., Conte, T.M.: Adaptive mode control: A static-power-efficient cache design. Trans. on Embedded Computing Sys. 2(3), 347–372 (2003)
25. Zhang, W., Kandemir, M., Karakoy, M., Chen, G.: Reducing data cache leakage energy using a compiler-based approach. Trans. on Embedded Computing Sys. 4(3), 652–678 (2005)

Author Index