Lecture Notes in Computer Science 3553

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Timo D. Hämäläinen   Andy D. Pimentel
Jarmo Takala   Stamatis Vassiliadis (Eds.)

Embedded Computer Systems: Architectures, Modeling, and Simulation

5th International Workshop, SAMOS 2005
Samos, Greece, July 18-20, 2005
Proceedings

Springer

Volume Editors

Timo D. Hämäläinen
Jarmo Takala
Tampere University of Technology
Institute of Digital and Computer Systems
Korkeakoulunkatu 1, 33720 Tampere, Finland
E-mail:{timo.d.hamalainen/jarmo.takala}@tut.fi

Andy D. Pimentel
University of Amsterdam
Department of Computer Science
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
E-mail: andy@science.uva.nl

Stamatis Vassiliadis
T.U. Delft, Computer Engineering
Mekelweg 4, 2628 CD Delft, The Netherlands
E-mail: s.vassiliadis@ewi.tudelft.nl

Library of Congress Control Number: 2005928160

CR Subject Classification (1998): C, B

ISSN 0302-9743
ISBN-10 3-540-26969-X Springer Berlin Heidelberg New York
ISBN-13 978-3-540-26969-4 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springeronline.com

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11512622 06/3142 5 4 3 2 1 0

Preface

The SAMOS workshop is an international gathering of highly qualified researchers from academia and industry, sharing in a 3-day lively discussion on the quiet and inspiring northern mountainside of the Mediterranean island of Samos. As a tradition, the workshop features workshop presentations in the morning, while after lunch all kinds of informal discussions and nut-cracking gatherings take place. The workshop is unique in the sense that not only solved research problems are presented and discussed but also (partly) unsolved problems and in-depth topical reviews can be unleashed in the scientific arena. Consequently, the workshop provides the participants with an environment where collaboration rather than competition is fostered.

The earlier workshops, SAMOS I–IV (2001–2004), were composed only of invited presentations. Due to increasing expressions of interest in the workshop, the Program Committee of SAMOS V decided to open the workshop for all submissions. As a result the SAMOS workshop gained an immediate popularity; a total of 114 submitted papers were received for evaluation. The papers came from 24 countries and regions: Austria (1), Belgium (2), Brazil (5), Canada (4), China (12), Cyprus (2), Czech Republic (1), Finland (15), France (6), Germany (8), Greece (5), Hong Kong (2), India (2), Iran (1), Korea (24), The Netherlands (7), Pakistan (1), Poland (2), Spain (2), Sweden (2), Taiwan (1), Turkey (2), UK (2), and USA (5). We are grateful to all of the authors who submitted papers to the workshop.

All the papers went through a rigorous reviewing process and, on average, each paper received three individual reviews. Due to time constraints in the workshop program and the high quality of the submitted papers, the selection process was very competitive and many qualified papers could not be accepted. Finally, the Program Committee selected 47 papers (corresponding to 41% of the submitted papers) for the workshop and the fifth edition of the SAMOS workshop developed into a highly interesting event. The program consisted of 32 plenary presentations (28% of the submitted papers), 15 poster presentations (13% of the submitted papers), and a keynote speech by Dr. Bob Iannucci from Nokia Research Center.

A workshop like this cannot be organized without the help of many other people. Therefore we want to thank the members of the Steering and Program Committees and the external referees for their dedication and diligence in selecting the technical presentations. The investment of their time and insight is very much appreciated. We would like to express our sincere gratitude to Timo D. Hämäläinen and Heikki Orsila for preparing the workshop proceedings, Stephen Wong for a successful publicity campaign, Elena Moscu Panainte and Iosif Antochi for maintaining the Web site and paper submission system, and Lidwina Tromp for her support in organizing the workshop.

We hope that the attendees enjoyed the SAMOS V workshop in all its aspects, including many informal discussions and gatherings.

June 2005

Andy Pimentel
Stamatis Vassiliadis
Jarmo Takala

Organization

The SAMOS V workshop took place during July 18–20, 2005 at the Research and Teaching Institute of East Aegean (INEAG) in Agios Konstantinos on the island of Samos, Greece.

General Chair

Andy Pimentel	University of Amsterdam, The Netherlands

Program Chair

Jarmo Takala	Tampere University of Technology, Finland

Proceedings Chair

Timo D. Hämäläinen	Tampere University of Technology, Finland

Publicity Chair

Stephen Wong	Delft University of Technology, The Netherlands

Steering Committee

Shuvra Bhattacharyya	University of Maryland, USA
Ed Deprettere	Leiden University, The Netherlands
Patrice Quinton	IRISA, France
Stamatis Vassiliadis	Delft University of Technology, The Netherlands
Jürgen Teich	University of Erlangen-Nuremberg, Germany

Program Committee

Koen Bertels	Delft University of Technology, The Netherlands
Luigi Carro	Universidade Federal do Rio Grande do Sul, Brazil
Nikitas Dimopoulos	University of Victoria, Canada
Pedro Diniz	University of Southern California, USA

Gerhard Fettweis	Technische Universität Dresden, Germany
Georgi Gaydadjiev	Delft University of Technology, The Netherlands
John Glossner	Sandbridge Technologies, USA
David Guevorkian	Nokia Research Center, Finland
Wayne Luk	Imperial College London, UK
Bernard Pottier	Université de Bretagne Occidentale, France
Tanguy Risset	IRISA/INRIA, France
Michael Schulte	University of Wisconsin-Madison, USA
Dirk Stroobandt	Ghent University, Belgium
Jarmo Takala	Tampere University of Technology, Finland
Serge Vernalde	IMEC, Belgium
Jens Peter Wittenburg	Thomson Corporate Research, Germany

Local Organizers

Lidwina Tromp	Delft University of Technology, The Netherlands
Yiasmin Kioulafa	Research and Training Institute of East Aegean, Greece

Referees

Antochi, I.
Bhattacharyya, S.
Bertels, K.
Bertels, P.
Blem, E.
Brune, T.
Calderon, H.
Carro, L.
Chang, Z.
Cheung, R.
Christiaens, M.
Crisu, D.
de Langen, P.
Devos, H.
Dimond, R.
Dimopoulos, N.
Diniz, P.
Duarte, F.
Erbas, C.
Faes, P.
Fettweis, G.
Fidjeland, A.
Gaedke, K.
Galuzzi, C.
Gaydadjiev, G.
Gehrke, W.
Glossner, J.
Guevorkian, D.
Guzma, V.
Han, X.
Hannig, F.
Heikkinen, J.
Heirman, W.
Hur, J.Y.
Hämäläinen, P.
Hämäläinen, T.
Iancu, D.
Jackowski, E.
Janes, D.
Jinturkar, S.
Järvinen, T.
Kachris, C.
Kangas, T.
Keinert, J.
Koch, D.
Kropp, H.
Kuorilehto, M.
Kuzmanov, G.
Langemeyer, S.
Langerwerf, J.M.
Lee, D.U.
Lehtoranta, O.
Li, B.
Li, S.
Luk, W.
Mamidi, S.
Martin, B.
Matus, E.
Mhamdi, L.
Molnos, A.
Moscu Panainte, E.
Moudgill, M.
Nacer, G.
Narkhede, P.
Orsila, H.
Pieper, S.
Pimentel, A.
Polstra, S.
Pottier, B.

Pourebrahimi, B.
Punkka, K.
Putzke-Röming, W.
Quinton, P.
Reuter, C.
Rissa, T.
Robelly, P.
Salmela, P.
Salminen, E.
Schulte, M.
Sedcole, P.
Senthilvelan, M.
Silvén, O.
Sima, M.
Smailbegovic, F.
Sourdis, I.
Stanek, Z.
Streichert, T.
Stroobandt, D.
Strydis, C.
Takala, J.
Teich, J.
Tsen, C.
Wang, L.-K.
Vassiliadis, S.
Vayá, G.P.
Winter, M.
Wittenburg, J.
Yli-Pietilä, T.
Yusuf, S.
Yuwono, I.

Table of Contents

Keynote

Reconfigurable System Design and Implementations

Processor Architectures, Design and Simulation

Architectures and Implementations

System Level Design, Modeling and Simulation

Platform Thinking in Embedded Systems

Bob Iannucci

Nokia Research Center,
Itämerenkatu 11-13, 00180 Helsinki, Finland

Abstract. Modern embedded systems are built from microprocessors, domain-specific hardware blocks, communication means, application-specific sensor/actuators and as simple as possible user interface, which hides the embedded complexity. The design of embedded systems is typically done in an integrated way with strong dependencies between these building block elements and between different parts of the system. This talk focuses on how platform thinking and engineering can be applied to increasingly complex embedded systems and what impacts that will have on the design and architectures. Platform engineering in embedded systems may sound contradictory, but in practice will introduce modularity and stable interfaces. New system-level architectures for hardware, middleware architectures, and certifiable operating system micro-kernels are needed to raise the abstraction level and productivity of design. As an example I will go through the definitions of some modules in a mobile device and the requirements for their interfaces. I will describe the additional design steps, new formal methods and system-level tasks that are needed in the platform approach. Finally, I will review the Advanced Research and Technology for Embedded and Intelligent Systems (ARTEMIS) technology platform in EU 7th Framework Program, which is bringing together industrial and academic groups to create coherent and integrated European research in the domain of embedded systems.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, p. 1, 2005.

Interprocedural Optimization for Dynamic Hardware Configurations

Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis

Computer Engineering Lab,
Delft University of Technology, The Netherlands
{E.Panainte, K.Bertels, S.Vassiliadis}@et.tudelft.nl

Abstract. Little research in compiler optimizations has been undertaken to eliminate or diminish the negative influence on performance of the huge reconfiguration latency of the available FPGA platforms. In this paper, we propose an interprocedural optimization that minimizes the number of executed hardware configuration instructions taking into account constraints such as the "FPGA-area placement conflicts" between the available hardware configurations. The proposed algorithm allows the anticipation of hardware configuration instructions up to the application's main procedure. The presented results show that our optimization produces a reduction of up to 3 - 5 order of magnitude of the number of executed hardware configuration instructions.

1 Introduction

The combination of a general purpose processor (GPP) and a Field Programmable Gate Array (FPGA) is becoming increasingly popular (e.g. [1], [2], [3], [4], [5] and [6]). Reconfigurable computing (RC) is a new style of computer architecture which allows the designer to combine the advantages of both hardware (speed) and software (flexibility). However, an important drawback of the RC paradigm is the huge reconfiguration latency of the actual FPGA platforms. As presented in [7], the potential speedup of the kernel hardware executions can be completely wasted by the repetitive hardware configurations that produce a performance decrease of up to 2 order of magnitude.

When targeting reconfigurable architectures, the compiler should be aware of the competition for the reconfigurable hardware resources (FPGA area) between multiple hardware operations during the application execution time. A new type of conflict - called in this paper "FPGA area placement conflict" - emerges between two hardware configurations that cannot coexist together on the target FPGA.

In this paper, we propose an interprocedural optimization that anticipates hardware configuration instructions up to the application's main procedure. The optimization takes into account constraints such as the "FPGA-area placement conflicts" between the available hardware configurations. The presented results show that a reduction of up to 3 - 5 order of magnitude of the number of executed hardware configuration instructions is expected for MPEG2 and M-JPEG multimedia applications.

This paper is organized as follows. Section 2 presents background information and related work for compiler optimizations targeting dynamic hardware configuration in-

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 2–11, 2005.

structions, followed by a motivational example in Section 3. The proposed interprocedural optimization algorithm is introduced in Section 4. Experimental results for two multimedia applications are provided in Section 5, and Section 6 presents the concluding remarks.

2 Background and Related Work

In this paper, we assume the Molen programming paradigm ([8], [9])which is a sequential consistency paradigm for programming Field-Programmable Custom Computing Machines(FCCMs) possibly including a general purpose computational engine(s). The paradigm allows for parallel and concurrent hardware execution and is intended (currently) for single program execution. It requires only a one time architectural extension of few instructions to provide a large user reconfigurable operation space. The added instructions include **SET** $< address >$ for reconfigurable hardware configuration and **EXECUTE** $< address >$ for controlling the executions of the operations on the reconfigurable hardware. In addition, two MOVE instructions for passing values to and from the GPP register file and the reconfigurable hardware are required.

In order to achieve significant performance improvement for real applications, more operations are usually executed on the reconfigurable hardware. As the available area of the reconfigurable platforms is limited, the coexistence of all hardware configurations on the FPGA for all application execution time may be restricted, resulting in ”FPGA-area placement conflicts”. Two hardware operations have an ”FPGA-area placement conflict” (or just conflict in the rest of the paper) if i) their combined reconfigurable hardware area is larger than the total FPGA area or ii) the intersection of their hardware areas is not empty.

Several aproaches have been proposed for reducing the impact of the reconfiguration latency on performance. A compiler approach that considers the restricted case of two consecutive and non-conflicting hardware operations is presented in [10]. In this approach, the hardware execution of the first operation is scheduled in parallel with the hardware configuration of the second operation. Our approach is more general as it performs scheduling for any number of hardware operations at procedural level and not only for two consecutive hardware operations. The performance gain produced by our scheduling algorithm results from reducing the number of performed hardware configurations. In [11], the reconfiguration overhead is reduced by using manual interprocedural optimizations such as localizing memory accesses, partial hardware reuse and pipeling. Our approach is different as the optimization is automatically applied in the compilation phase and it minimizes the number of performed hardware configurations without specific information about the target FPGA and hardware operations. The instruction scheduling approach presented in [12] uses data-flow analyses and profile information for reducing the number of executed hardware configurations. In this paper, we extend this approach at interprocedural level, taking into account all procedures of the target applications. As a consequence, the impact of the proposed optimization is significantly increased as presented in Section 5.

3 Motivation and Contribution

In order to illustrate the goals and the main features of the proposed interprocedural optimization, we present in Fig. 1 a motivational real example. The presented subgraph is included in the call graph of the MPEG2 encoder multimedia benchmark where an edge $< p_i, p_j >$ represents a call from procedure p_i to procedure p_j. We consider that the procedures SAD, DCT and IDCT are executed on the reconfigurable hardware and that initially the hardware configuration (a SET instruction) is performed before each hardware execution (an EXEC instruction). One first observation is that the configuration for the SAD operation can be safely anticipated in the **motion_estimation** procedure. This anticipation will significantly reduce the number of performed hardware configurations as it will not be performed for each macroblock but only for each frame of the input sequence. This observation also holds for the DCT configuration in **transform** and the IDCT configuration in **itransform**. Moreover, the SAD configuration from **motion_estimation** can be moved upwards in the **putseq** procedure, immediately preceding the call site of **motion_estimation** in **putseq**. Additionally, it can be noticed that the propagation of the SAD configuration from **putseq** to the **main** procedure depends on the FPGA area allocation for SAD, DCT and IDCT. When the SAD operation does not have any FPGA-area placement conflict with the other two hardware operations DCT and IDCT, its configuration can be safely performed only once, at the entry point in the **main** procedure.

The contribution of this paper includes the following. The optimization proposed in this paper allows to anticipate the hardware configurations at interprocedural level, while prior work was limited to optimizations at procedural level (intraprocedural).

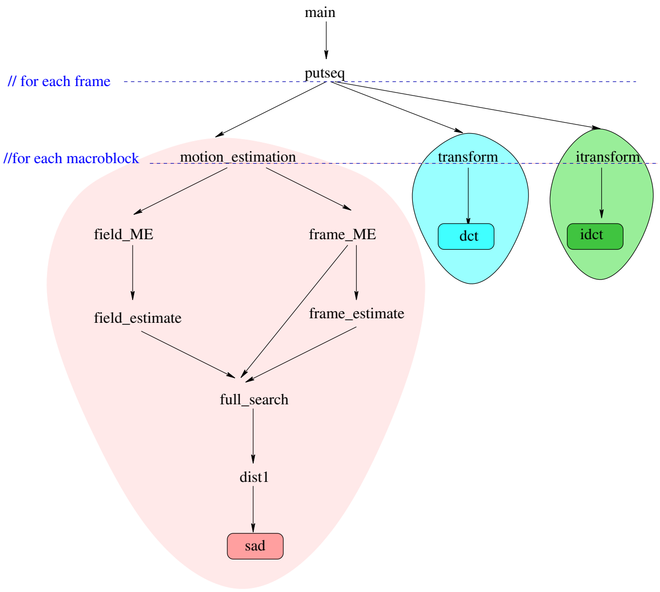


Fig. 1. Motivational example for MPEG2 encoder

Secondly, although the interprocedural optimizations are considered to provide little benefit and significantly increase the compiler complexity, we show that our optimization significantly reduces the number of hardware configurations (a major drawback of the current FPGAs).

4 Interprocedural Optimization for Dynamic Hardware Configurations

The main goal of the proposed interprocedural optimization presented in this section is to anticipate the dynamic hardware configuration instructions taking into account the hardware conflicts between the available hardware operations. As such hardware configuration does not cause an exception, a speculative algorithm is used for anticipating the hardware configuration instructions. The interprocedural optimization consists of three steps. In the first step, the program's call graph is constructed based on an interprocedural control-flow analysis. Next, the set of live hardware configurations for each procedure is determined using an interprocedural data-flow analysis. Finally, the hardware configuration instructions are anticipated in the call graph taking into account the available conflicting operations.

4.1 Step 1: Interprocedural Control-Flow Analysis for Dynamic Hardware Configurations

Starting point of the proposed optimization is the construction of the program's call graph. Given a program P consisting of a set of procedures $< p_1, p_2, ..., p_n >$, the program's call graph of P is the graph $G =< N, E, r >$ with the node set $N = \{p_1, p_2, ..., p_n\}$, the set $E \subseteq N$ x N, where $< p_i, p_j > \in E$ denotes a call site in p_i from which p_j is called, and the distinguished entry node $r \in N$ representing the main entry procedure of the program . An example of a real call (sub)graph is presented in Fig. 1.

The construction of the call graph for a program written in C is straightforward as there are no higher-order procedures in the C programming language. For this purpose, we used the *sbrowser_cg* library included in the *suifbrowser* package available in the SUIF environment. The constructed call graph is the input of the optimization algorithm presented in Table 1. As explained in the next subsection, the constructed graph is required to be a DAG (Directed Acyclic Graph) (see Table 1, step 1).

4.2 Step 2: Interprocedural Data-Flow Analysis for Dynamic Hardware Configurations

The goal of the interprocedural data-flow analysis is to determine what hardware operation can modify the FPGA configuration as a side effect of a procedure call. We define LRMOD(p) (Local Reconfigurable hardware MODification) as the set of hardware operations associated with a procedure p. In order to simplify this discussion, we assume that there is at most one hardware operation that can be associated with a procedure. More specifically, $op_1 \in LRMOD(p)$ if there is a pragma annotation that indicates that procedure p is executed on the reconfigurable hardware and its associated hardware operation is named op_1. $RMOD(p)$, Reconfigurable hardware MODification, represents

Table 1. The interprocedural optimization algorithm for hardware configuration instructions

```
Interprocedural Optimization Algorithm

INPUT: Call graph $G =< N,S,r >$, hardware conflicts $f : HWxHW- > \{0,1\}$
OUTPUT: Insertion edges L

1. //Verify assumptions for G
    check if G is DAG

2. //RMOD computation
   traverse G in reverse topological order
        compute for each procedure p
        $RMOD(p) = LRMOD(p) \bigcup_{s \in Succ(p)} RMOD(s)$

 //Compute CF
    for each procedure p
        $CF(p) = \{op_1 \in RMOD(p) | \exists op_2 \in RMOD(p), op_1 \nleftrightarrow op2\}$
3. //Compute the insertion edges
    $L = \emptyset$
    for each edge $< p_i, p_j >$
        for each $op \in [RMOD(p_j) - CF(p_j)] \cap CF(p_i)$
            $L = L \cup < p_i, p_j, op >$
    for each $op \in [RMOD(r) - CF(r)]$
        $L = L \cup < r, r, op >$
```

the set of all hardware operations that may be executed by an invocation of procedure *p* and it can be computed using the following data-flow equation:

$$RMOD(p) = LRMOD(p) \bigcup_{s \in Succ(p)} RMOD(s) \tag{1}$$

A hardware operation *op* may be performed by calling procedure *p* if *op* is associated with procedure *p* (i.e. $op \in LRMOD(p)$) or if it can be performed by a procedure that is called from procedure *p*. For an efficient computation, the RMOD values should be computed in reverse topological order (i.e. reverse invocation order) when the call graph does not contain cycles (see step 2 from Table 1). The RMOD values for the example presented in Fig. 1 are shown in Fig. 2. For the basic blocks where LRMOD values are missing, they are implicitly assumed as $\emptyset$. We notice that by calling *putseq* procedures, all three hardware operations sad, dct and *idct* may be executed on the reconfigurable hardware.

Due to the increasing complexity of the interprocedural data-flow analysis, this step is performed only when the call graph G satisfies the following criteria. We assume that there are no *indirect procedure calls* (using pointer to functions). These limitations can be eliminated by considering all candidate set of functions that have the same prototype. Another limitation concerns the data-flow equations for procedures with recursive procedure calls (when the call graph contains cycles). In this case, the strongly connected

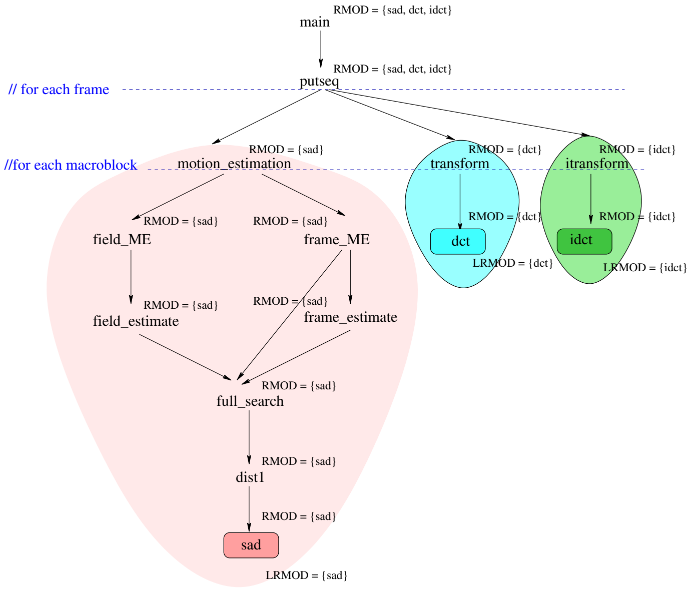


Fig. 2. Interprocedural data-flow analysis for MPEG2 encoder

components (scc) should be computed and the data-flow equations should be collapsed for each scc into a single equation. The proposed optimization is applied only when the call graph is a DAG.

4.3 Step 3: Interprocedural Scheduling for Dynamic Hardware Configuration Instructions

In this step, the hardware configuration instructions are anticipated in the call graph taking into account the possible hardware conflicts discovered in the previous step. In the first phase, the set of conflicting operations $CF(p)$ is computed for each procedure included in the call graph based on the $RMOD$ values as follows:

$$CF(p) = \{op_1 \in RMOD(p) | \exists op_2 \in RMOD(p), op_1 \nleftrightarrow op2\} \tag{2}$$

Next, for each edge of the call graph $< p_i, p_j >$, if there is an hardware operation op which does not have conflicts in p_j ($op \not\in CF(p_j)$) but it has conflicts in the calling function p_i ($op \in CF(p_i)$), then a SET op instruction is inserted at all call sites of p_j from p_i. Finally, for all non-conflicting operations of the entry node of the call graph G (i.e. $RMOD(r) - CF(r)$), the corresponding SET instructions are inserted at the beginning of the r procedure (see step 3 from Table 1).

The CF values for the example presented in Fig. 1 are shown in Fig. 3, for the case where all considered hardware operations conflict with each other. For the basic blocks where CF values are missing they are implicitly assumed as $\emptyset$. It can be noticed that the hardware configuration instructions cannot simultaneously propagate upwards of *putseq* procedure due to the considered hardware conflicts.

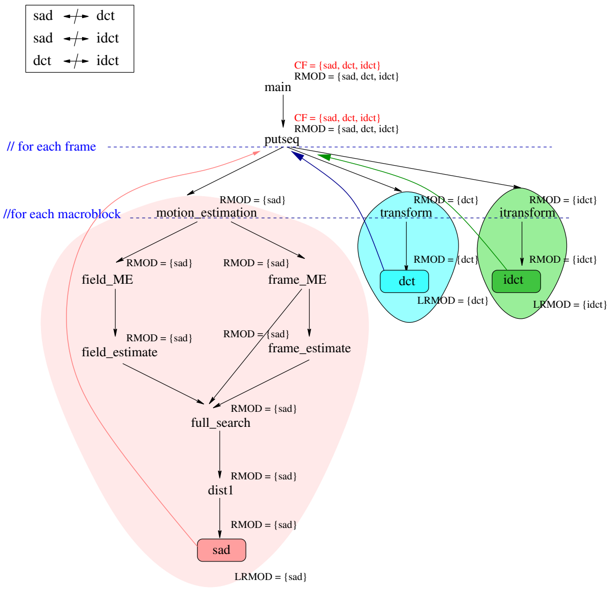


Fig. 3. Interprocedural optimization for MPEG2 encoder

5 Results

In order to present the results of our algorithm with respect to the number of performed hardware configurations, we first describe the experimental setup, including the target architecture and applications. Next, we concentrate on the impact of the optimization on the number of hardware configurations. Finally we present several important observations about the presented results and possible improvements of the optimization algorithm.

5.1 Experimental Setup

The application domain of these experiments is video data compressing. We consider two real-life applications namely Motion JPEG (M-JPEG) encoder which compresses sequences of video frames applying JPEG compression for each frame and the MPEG2 encoder multimedia benchmark. The input test sequence for M-JPEG contains 30 color frames from ”tennis” in YUV format with a resolution of 256x256 pixels. For MPEG2 encoder, we used the standard test frames included in the benchmark. The operations considered for execution on the FPGA for the M-JPEG applications are DCT (2-D Discrete Cosine Transform), Quantization and VLC (Variable Length Coding), while for MPEG2 the candidates are SAD (Sum of Absolute-Difference), DCT and IDCT (inverse DCT).

The described optimization algorithm has been implemented in the Molen compiler, more specifically in the SUIF compiler frontend. We used the *link_suif* pass that combines all input SUIF files, making their global symbol tables consistent and the *sbrowser_cg* library included in the *suifbrowser* package available in the SUIF environment for the construction of the interprocedural call graph. The call graph for the M-JPEG encoder includes 47 nodes (i.e. the applications contains 47 procedures), while the call graph for MPEG2 encoder (a subgraph is presented in Fig. 1) has 111 nodes.

5.2 Interprocedural Optimization Results

The aim of the proposed optimization is to significantly reduce the number of the executed SET instructions for each hardware operation. In the results presented in the rest of this section, we compare the number of executed hardware configurations with and without our optimization (denoted as SET_OPT and respectively NO_SET_OP cases).

M-JPEG Encoder Results. Table 2 shows the number of hardware configurations required in the M-JPEG encoder multimedia application for the SET_OPT (columns 3-7) and NO_SET_OPT (column 2) cases. When measuring the effects of the proposed optimization (Table 2, columns 3-7), we consider different possible conflicts between DCT, Quant and VLC; in the best case there is no conflict (column 3), while in the worst case all hardware operations are in conflict with each other (column 7). The first observation is that, for the *no conflict* case, our optimization algorithm eliminates all hardware configurations and introduces at the application entry point only one hardware configuration for each hardware operation; thus, all the hardware configurations but one from the initial application (Table 2, column 2) are redundant. A second observation is that our optimization reduces the number of DCT hardware configurations with at least 75 % for all conflict cases. Finally, we notice that even for the worst case (Table 2, columns 7), the proposed optimization reduces the number of executed SET instructions for DCT configuration by 4x. This reduction is due to the anticipation of DCT hardware configuration at the macroblock level, while the configurations for Quant and VLC are already performed at this level and cannot be anticipated upwards due to the hardware conflicts.

MPEG2 Encoder Results. The number of hardware configurations for the considered functions in the MPEG2 encoder benchmark is presented in Table 5.2. One important observation is the 3-5 order of magnitude decrease of the number of hardware configurations produced by our optimization algorithm for all conflict cases. The main cause of this decrease is the particular features of the MPEG2 algorithm where the SAD, DCT

Table 2. The impact of the interprocedural optimization on the number of required hardware configurations in M-JPEG encoder

	Initial	With interprocedural SET optimization				
HW op	[# SETs]	No conflict	DCT Quant conflict	DCT VLC conflict	Quant VLC conflict	DCT Quant VLC conflict
DCT	61440	1	15360	15360	1	15360
Quant	15360	1	15360	1	15360	15360
VLC	15360	1	1	15360	15360	15360

Table 3. The impact of the interprocedural optimization on the number of required hardware configurations in MPEG2 encoder

	Initial	With interprocedural SET optimization				
HW op	[# SETs]	No conflict	SAD DCT conflict	SAD IDCT conflict	DCT IDCT conflict	SAD DCT IDCT conflict
SAD	117084	1	3	3	1	3
DCT	1152	1	3	1	3	3
IDCT	1152	1	1	3	3	3

and IDCT hardware configurations can be anticipated out to the frame level rather than macroblock level (see Fig. 3). In consequence, due to our optimization algorithm, the hardware configuration is transformed from a major bottleneck in a negligible factor on performance.

In order to conclude this section, four points should be noticed regarding the presented results and optimization. Firstly, the reduction of the number of hardware configurations depends on the characteristics of the target applications. As previously presented, the impact of our optimizations for MPEG2 encoder is substantial, while for other applications (e.g. M-JPEG) it depends on the possible hardware conflicts between operations. Second, it should be mentioned that this optimization can also increase the number of hardware configurations, e.g. when the considered procedure associated to the hardware operations have multiple call sites and conflicting operations. Flow-sensitive data-flow analysis and profile information can be used to prevent this situation. Nevertheless, taking into account that the hardware configuration can be performed in parallel with the execution of other instructions on the GPP, the reconfiguration latency may be (partially) hidden. The final major point is that a significant reduction of the number of executed hardware configurations is directly reflected in a significant reduction in power consumption, as the FPGA reconfigurations is a main source of power consumption.

6 Conclusions

In this paper, we have proposed an interprocedural optimization algorithm for hardware configuration instructions. This algorithm takes into account specific features of the target applications and of the reconfigurable hardware such as the ”FPGA area placement conflicts”. It allows the anticipation of hardware configuration instructions up to the application’s main procedure. The presented results show that our optimization produces a reduction of up to 3 - 5 order of magnitude of the number of executed hardware configuration instructions for the MPEG2 and M-JPEG multimedia benchmarks.

Future research will focus on compiler optimizations to allow for concurrent execution. We also intend to extend the compiler to provide information for an efficient FPGA area allocation of the different hardware operations in order to eliminate the FPGA-area placement conflicts.

References

1. Campi, F., Cappelli, A., Guerrieri, R., Lodi, A., Toma, M., Rosa, A.L., Lavagno, L., Passerone, C.: A reconfigurable processor architecture and software development environment for embedded systems. In: Proceedings of Parallel and Distributed Processing Symposium, Nice, France (2003) 171–178
2. Sima, M., Vassiliadis, S., S.Cotofana, van Eijndhoven, J., Vissers, K.: Field-Programmable Custom Computing Machines - A Taxonomy. In: 12th International Conference on Field Programmable Logic and Applications (FPL). Volume 2438., Montpellier, France, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2002) 79–88
3. Becker, J.: Configurable Systems-on-Chip : Commercial and Academic Approaches. In: Proc. of 9th IEEE Int. Conf. on Electronic Circuits and Systems - ICECS 2002, Dubrovnik, Croatia (2002) 809–812
4. Gokhale, M.B., Stone, J.M.: Napa C: Compiling for a Hybrid RISC/FPGA Architecture. In: Proceedings of FCCM'98, Napa Valley, CA (1998) 126–137
5. Rosa, A.L., Lavagno, L., Passerone, C.: Hardware/Software Design Space Exploration for a Reconfigurable Processor. In: Proc. of DATE 2003, Munich, Germany (2003) 570–575
6. Ye, Z.A., Shenoy, N., Banerjee, P.: A C Compiler for a Processor with a Reconfigurable Functional Unit. In: ACM/SIGDA Symposium on FPGAs, Monterey, California, USA (2000) 95–100
7. Moscu Panainte, E., Bertels, K., Vassiliadis, S.: Dynamic hardware reconfigurations: Performance impact on mpeg2. In: Proceedings of SAMOS. Volume 3133., Samos, Greece, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2004) 284–292
8. Vassiliadis, S., Gaydadjiev, G., Bertels, K., Moscu Panainte, E.: The Molen Programming Paradigm. In: Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation, Samos, Greece (2003) 1–7
9. Vassiliadis, S., Wong, S., Gaydadjiev, G.N., Bertels, K., Kuzmanov, G., Moscu Panainte, E.: The Molen Polymorphic Processor. IEEE Transactions on Computers **53(11)** (2004) 1363–1375
10. Tang, X., Aalsma, M., Jou, R.: A Compiler Directed Approach to Hiding Confguration Latency in Chameleon Processors. In: FPL. Volume 1896., Villach, Austria, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2000) 29–38
11. Mei, B., Vernalde, S., De Man, H., Lauwereins, R.: Design and Optimization of Dynamically Reconfigurable Embedded Systems. In: Proceedings of Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas, Nevada, USA (2001) 78–84
12. Moscu Panainte, E., Bertels, K., Vassiliadis, S.: Instruction Scheduling for Dynamic Hardware Configurations. In: Proceedings of Design, Automation and Test in Europe 2005 (DATE 05), Munich, Germany (2005) 100–105

Reconfigurable Embedded Systems: An Application-Oriented Perspective on Architectures and Design Techniques

M. Glesner, H. Hinkelmann, T. Hollstein, L.S. Indrusiak, T. Murgan,
A.M. Obeid, M. Petrov, T. Pionteck, and P. Zipf

Institute of Microelectronic Systems, Darmstadt University of Technology,
D-64283 Karlstr. 15, Darmstadt, Germany
glesner@mes.tu-darmstadt.de

Abstract. Reconfiguration emerged as a key concept to cope with constraints regarding performance, power consumption, design time and costs posed by the growing diversity of application domains. This work gives an overview of several relevant reconfigurable architectures and design techniques developed by the authors in different projects and emphasizes the effective role of reconfigurability in embedded system design.

1 Introduction

Embedded systems have specific requirements and perform tasks, which must run generally with power consumption and real-time operation constraints while keeping design and maintenance costs low. By utilizing run-time adaptable hardware, reconfigurable architectures offer a trade-off between the performance of ASICs and the flexibility of less power efficient general purpose processor.

Reconfigurability offers several key advantages. *Functionality on demand:* hardware functionality can be changed or optimized after system deployment. *Acceleration on demand:* certain applications can be accelerated by customizing data-paths and operators on a massive parallel scale. *Shorter time-to-market:* hardware configuration and software can be developed in parallel, which implies great flexibility to late design changes and bug fixes. *Extended product life-cycles:* as flexibility is preserved, manufactured devices can be adapted to standard or customer specifications not considered at design time. *Low design & maintenance costs:* functionality adaptation (specialization, upgrade, acceleration), reduced development time and extended life-cycle imply low design and maintenance costs, the most salient feature of reconfigurable architectures.

Based on a set of industry and academia relevant applications, this paper gives an overview of reconfigurable architectures for embedded systems and the design thereof. The paper tries by no means to be exhaustive as the main focus lies on architectures and methodologies developed by the authors in different projects. The paper is organized as follows: section 2 presents three case studies applying reconfigurable architectures for multi-functional support, dynamic power-performance management, and DSP-specific architectural optimization. Section 3 discusses system integration issues for reconfigurable systems at different abstraction levels. Section 4 describes methodologies for reconfigurable systems design and validation. The paper concludes with some final remarks.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 12–21, 2005.

2 Applications and Architectures

Functional Optimization and Multi-functional Support. Reconfigurable platforms allow embedded devices to be upgraded, optimized or modified after deployment to final protocol standards or to architectural and algorithmic solutions which were not even considered at design time. In order to reduce design costs and risks, designers aiming for an early presence on the market focus more and more on flexible architectural solutions. A typical application scenario where the standardization process spans a long period of time are optical transport networks (OTN). In order to reduce re-spin costs, we proposed a multiple-core adaptive extension for overhead processing to a multi-rate forward error correction code device developed by Lucent Technologies [1].

As represented in Fig. 1, a node processing block is built around the isolated Integer Unit (IU) of the Leon2 processor. Incoming data blocks are sent to each node through the Source Bus and a Dual-Port RAM. Similarly, after the data is processed, the nodes send the resulting data packets through the Sink Bus. A run-time functional modification is realized by using the double program memory of each Boot Bus Bridge (BBB), as processor cores can switch between the two program memories. A system update implies that modified code is loaded into the boot memory and afterwards into the BBB. Multi-standard support can be implemented in a similar fashion. Additionally, the nodes can also communicate via a Shared Memory and the isolated IUs can be extended with dedicated or reconfigurable logic for performance improvement [1].

Dynamic Power-Performance Management. Architecture optimization for low-power can be either static, assuming a worst case scenario at design-time, or dynamic, when the architecture needs to be extended to support run-time reconfigurability. In our research, we have chosen the Viterbi decoder as a case study, as it accounts for a significant percentage of the power consumption in digital receivers. The main contributor is the survivor management block, whose power consumption increases linearly with the length, which is a parameter of the design and directly affects the performance. By dynamically adjusting this length according to the run-time variations in the channel quality, significant power savings can be achieved.

Within the scope of our research we have developed an architecture for the trace-back unit that allows its length to be adjusted dynamically [2], thus saving power by

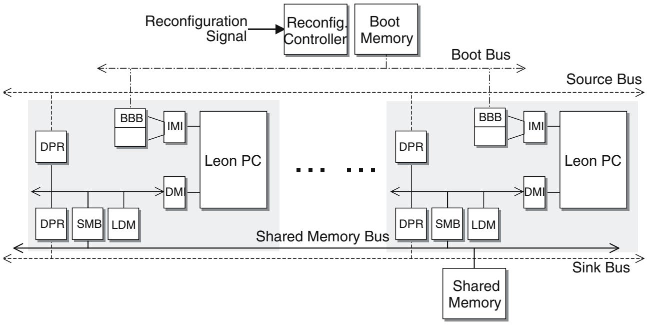


Fig. 1. Multiple-Core Architecture for Overhead Processing

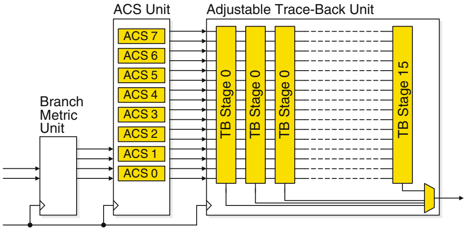


Fig. 2. Viterbi Decoder with Adjustable Trace-Back

keeping only the active part on, together with an algorithm for determining the optimal trace-back window length which ensures a target bit error rate of the decoded output. In order to determine the BER experimentally as a function of different factors, a highly parameterizable simulation chain has been created using SystemC. It allows a sweep analysis of the *BER* as a function of two variables: trace-back window length and the *SNR* of the channel. The architecture consisting out of Branch Metric Unit (BMU), Add-Compare-Select (ACS), and Trace-Back (TB) Unit is depicted in Fig. 2, where the trace-back has been pipelined into 16 stages [3]. When lower performance is required, the unused stages are disabled, thus saving power. Determining the optimum length for a given SNR and puncturing pattern is done externally by an embedded processor. We applied the principle of dynamic power reduction to a state-parallel Viterbi decoder for the IEEE 802.11a standard implemented in a 0.13um CMOS library and demonstrated power savings of up to 62% [3].

Architectural Optimization for Specific DSP Applications. One of the main problems with reconfigurable solutions is the overhead area and power consumed by reconfiguration resources. Reconfiguring vectors rather than bits and thus achieving notable area savings, Coarse-Grained Reconfigurable Architectures (*CGRA*) have attracted lots of attention in both research and industry communities. In order to achieve further savings in reconfiguration resources a pragmatic solution is to focus on the DSP algorithms to be realized by the CGRA. By studying the data-flow graphs of these algorithms, common features can be extracted and specific design decisions can be taken, tailoring the CGRA to the DSP applications of interest.

In the context of our research, we have studied several DSP algorithms including filtering, FFT, DCT, and Viterbi decoding. We started out by studying their data flow graphs, and their reported VLSI implementations. Thereafter, parameterizable VHDL models were developed and tested [4, 5]. This paved the way for realizing a CGRA that can efficiently solve these DSP algorithms.

Our approach (simplified in Fig. 3) is to use a Hybrid CGRA of processing elements (*PE*s) of different types. Spreading the functionality on different types of PEs helps in realizing a more area-efficient reconfigurable array, yet maintaining the required functionality. Two types of logic and arithmetic PEs along with memory manipulations PE are used. A configuration and operation controller orchestrates the operation and reconfigurations of the Hybrid CGRA Block.

The issue of dynamic reconfiguration is an important one to address not only because of the advantages and opportunities that can be visited by having such a capability,

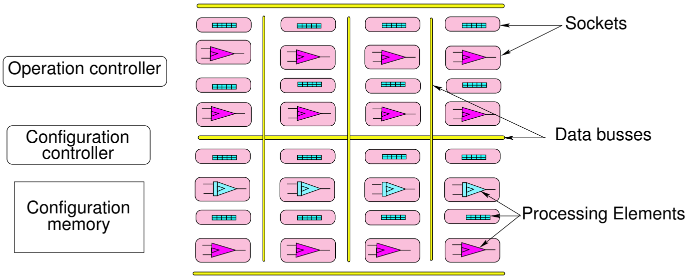


Fig. 3. Hybrid Reconfigurable DSP Architecture

but moreover to facilitate the implementation of some techniques that require dynamic reconfiguration of the data path on the fly, take the Radix 2 Single Delay Feedback architecture (R2SDF) as an example [6]. To address this problem we propose to solve the dynamic reconfiguration problem in two parts: locally and globally. Locally: all PEs as well as interconnection resources are placed in sockets having a set of configuration bits stored in local registers. Globally: the operation controller selects the configuration of the PE from the set stored in the socket and the configuration controller updates the local configuration bits in the sockets.

3 System Integration

Processor Integration of Reconfigurable Architectures. In a typical embedded systems scenario, a regular processor takes control over the system and the application, while reconfigurable hardware is applied to accelerate certain computation-intensive tasks. In such systems the question is how to couple processors and reconfigurable architectures. A common classification is to distinguish between the direct integration of a reconfigurable architecture into the data-path of a processor (i.e. as a reconfigurable functional unit = RFU), its connection as a coprocessor or its attachment as a peripheral device via an I/O bus. A description of advantages and disadvantages of each class can be found in [7], as well as various references to architecture examples. RFUs represent the tightest form of coupling and therefore reduce communication costs with the processor to a minimum, making it possible to map efficiently even smallest tasks to the RFU. However, as pointed out in [8], several drawbacks and severe integration problems have to be considered, e.g. the lack of a separate memory interface for the RFU or pipeline conflicts that have to be solved at run-time. Contrary to this, coprocessors and peripherals are easier to integrate and require less processor control, but communication costs are higher. Thus, they are better suited to execute larger tasks that can run independently from the processor.

In [7] and [8] we propose a solution for integrating an RFU into a complex RISC processor that combines advantages of all three integration possibilities and is shortly presented in the following. The respective reconfigurable architecture has been developed during our research on reconfigurable computing systems for the MAC layer of

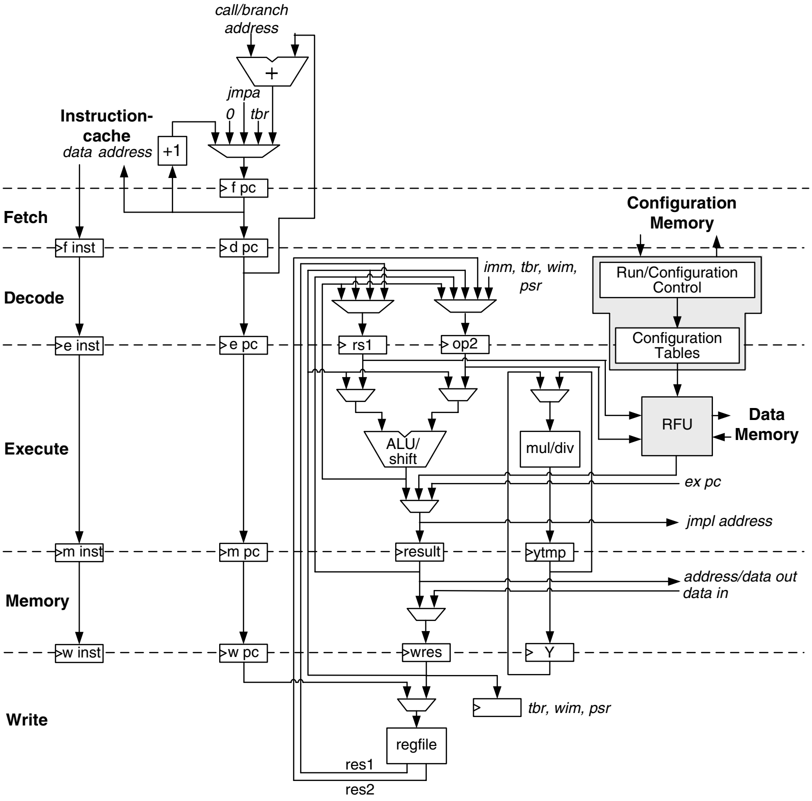


Fig. 4. Pipeline of the LEON2 processor with integrated RFU

WLANs and is capable to perform error detection/correction and cryptographic operations. It has been integrated as an RFU into the 5-stage instruction pipeline of the LEON2 processor, which had to be modified appropriately as shown in Fig. 4. To avoid pipeline conflicts, additional hold signals have been inserted that enable the RFU and the processor to coordinate and to stall each other on demand [8]. The instruction set of the LEON2 has been extended by three RFU specific instructions: two for initiating RFU operations (lasting one or multiple cycles) and one for reconfiguring the RFU. Inside the RFU, these instructions are decoded and processed by a novel configuration and run control unit [7]. It controls reconfiguration and data processing operations independently from the processor, thus significantly reducing the related overhead. With the help of several interacting configuration tables and a dedicated configuration memory, it is possible to dynamically reconfigure the device every clock cycle. The RFU has also been extended with a memory interface and dedicated control circuitry, allowing it to efficiently process large data blocks while the processor can work in parallel on different tasks. Simulation and synthesis results prove our concept and show a significant performance gain [8].

NoC-Enabled System-Level Reconfiguration. System-on-Chip (SoC) integration of multiple components, which are reconfigurable at IP block level (embedded FPGA or processor cores), poses new challenges concerning system reconfigurability. In this context, several scenarios can be assumed: reconfigurable SoC platforms used for customizing different product derivatives based on the same underlying SoC hardware (static configuration, firmware updates reconfiguration); flexible hardware platforms performing several tasks in parallel with dynamic resource allocation (dynamic reconfiguration, time-dependent task assignments). System-level reconfiguration can be understood as the reconfiguration of the implemented functionality of one or multiple SoC IP cores (FPGAs, processors). In many cases such a reconfiguration will end up in a modified communication profile within the overall SoC on-chip communication. In order to be able to provide the required communication flexibility, the on-chip interconnection architecture must provide a dynamic backbone, which can provide the required communication modes and sufficient bandwidth. Networks-on-Chip (NoCs) provide a new interconnection paradigm: abstraction of the real communication architecture as NoC-attached components can communicate by using virtual addresses. Several proposals for SoC architectures have been presented:[9] (Philips), NOSTRUM [10], SOCBUS, PROTEO, SPIN, HiNoC [11], to cite only a few.

Increased difficulties for clock distribution in large SoCs, imply the requirement of asynchronous communication at the top level, which leads to the so-called Globally Asynchronous and Locally Synchronous (GALS) design style.The proposed *HiNoC*[12] system for on-chip communication supports those designs and offers support for dynamic system-level reconfiguration. Moreover, it provides both packet-based (best-effort) and stream-based (with guaranteed Quality-of-Service) communication services.

In contrast to other approaches, *HiNoC* provides a top-level mesh-based topology and on the second level within synchronous domains a fast FAT-tree based network (two-level topology). An essential property, which makes *HiNoC* unique is the inherent support of heterogenous topologies. By combining static and dynamic routing, dynamically changing architectures can be realized.

Figure 5 illustrates two operational scenarios in one NoC-based system. In the first scenario, processes $P1$ to $P5$ are running. In contrast to that, $P4$ and $P5$ are not running in the second scenario. This could be because the task requiring those two processes has finished. Since the associated hardware resources are unused, the network can be partly shut down if the remaining links can fulfill the communication requirements of the system in the new configuration. The right part of Fig. 5 depicts the architectural changes to the system. The *HiNoC* Communication Architecture provides a time-division multiplex access scheme which is combined with a wormhole routing method (possibility

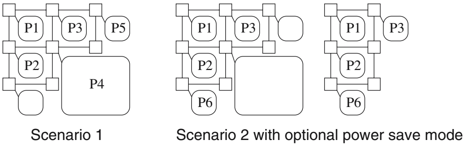


Fig. 5. HiNoC: Operational Scenarios

of folding packets and streams). Connections with guaranteed bandwidth (QoS) are set up by a connection request packet.

The main problem of data transfers between truly independent clock domains is the possibility of metastability in the receiving flip-flops. To avoid metastability in such asynchronous communication fast and robust synchronization techniques must be employed. In a conventional synchronous design, each bit of transferred data is assumed to become stable long enough before the next clock edge. Data transfers can thus be implemented by directly connecting the output of the sending module to the input registers of the receiving module. After the incoming data became stable, it is written into the registers with the rising edge of the clock signal. In a globally asynchronous system, there is no dependency of the clocking of an individual module on another. Because of this missing dependency, no assumption can be made on the time difference between the transition of an incoming data or control signal and the next local clock edge. If both transitions appear at the receiving flip-flop within a time-frame small enough (in terms of its setup and hold times), then the flip-flop can become metastable.

Several approaches for synchronization have been proposed which can be applied to NoC switch communication, including FIFO buffers [13], self-synchronization [14], or adaptations of clock stretching [15]. Our own experiments with NoC switch designs indicate that it is possible to obtain save and efficient implementations [16] for the asynchronous switch communication.

4 Methodologies for System Design and Validation

Reconfigurable systems inherit most of the design complexity of regular hardware/software systems. Thus, the design methodologies for reconfigurable systems are also based on model refinement across several abstraction layers. The added complexity of designing and validating reconfiguration mechanisms is currently addressed in the lower abstraction layers. For FPGA based systems, a common approach is the Modular Design Flow, which advocates that different configurations should be designed separately at RTL after an initial floor-plan budgeting [17]. While such low-level methods for reconfiguration are necessary, we advocate here a system-level view in order to actually validate the benefits of reconfigurability within an application scenario. Such scenarios are usually modeled as complex testbenches that are unpractical in RTL, and the accurate modeling of their dynamic nature is particularly critical in the case of run-time reconfiguration. Thus, designers must first design the system and its testbench in a high abstraction level, in order to allow for design space exploration, and in a second step the system must be designed in lower levels - logic or RTL - so that its implementation-specific features can be defined. In order to validate the low-level implementation model, its co-simulation with the high-level testbench (or at least part of it) should be possible.

The approach presented here to cope with such demands is strongly based on the actor-oriented design methodology presented by Lee et al [18]. We explore the concept of actors to model both the reconfigurable systems and the application scenarios where they are inserted, allowing for a well defined strategy for system modularization. Once the actors are well defined, it is possible to refine each one of them individually towards implementation models in logical or RTL level. It is also possible to devise mechanisms to allow the simulation of the implementation models with the remaining actors by encapsulating them (together with their respective simulators, when possible) as actors.

As a case study, we applied such approach on the design of a self-reconfigurable receiver for WCDMA down-link UMTS standard. This receiver can self-adapt to the conditions of the transmission channel due to multi-path [19], combining the conventional Rake receiver with maximal ratio combining (MRC) and a linear minimum mean squared error (LMMSE) equalizer. We first explored the actor-oriented modeling capabilities of Ptolemy II [20] by creating models of both the receiver and its usage scenario including user mobility patterns and transmission channel conditions. The accurate description of the application scenario increases significantly the complexity of the model, therefore it is implemented in a high level of abstraction using floating point arithmetic. Figure 6 shows PtolemyII GUI displaying the top-level model of the communication system. The composite actors *RakeReceiver* and *Equalizer* are also shown in a lower hierarchical level as those are the ones to be implemented in hardware. By simulating the system in PtolemyII, designers can navigate over the design space by parameterizing the receiver and analyzing the bit error rate (BER) under different channel models. The implemented case study used the channel described in [20].

After the analysis and profiling of the system model, the hardware implementation of the receiver was started: Rake receiver, MRC and the MMSE linear equalizer. We used JHDL [21] to create lower abstraction models of the subsystems that should be implemented in hardware. JHDL is a HDL based in Java and was chosen for the simplicity of integration with PtolemyII, as shown in [22], but there is no restriction on this point and the methodology would be the same if other HDL such as SystemC or VHDL were used. While some early conceptual work is shown in [18], no automated path from PtolemyII models to JHDL descriptions are available, thus the implementation is based on iterative refinement. In order to validate the refinement steps, every module written in JHDL should be encapsulated within an actor, integrated into Ptole-

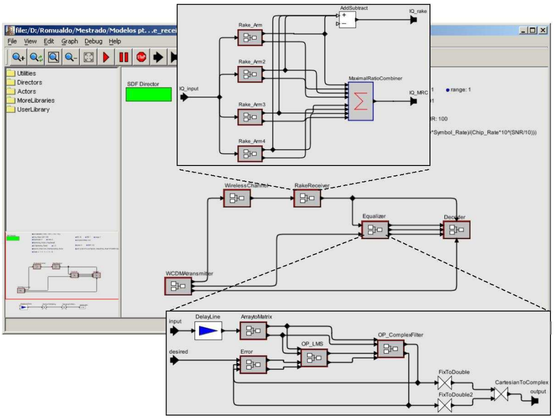


Fig. 6. Top-level system view in PtolemyII

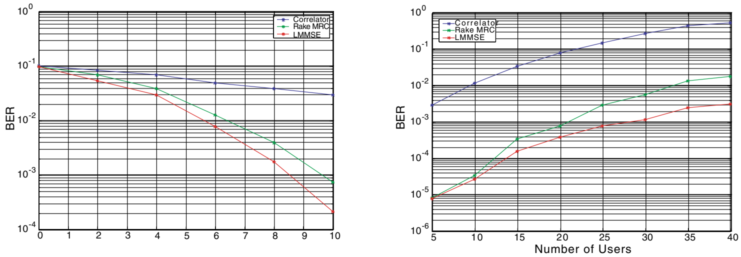


Fig. 7. a) BER versus SNR for receiver in a vehicular channel with 120Km/h, spreading factor 64, and 15 users; b) BER versus number of users for a receiver in a vehicular channel with 60Km/h, spreading factor 64 and SNR 10dB

myII and co-simulated together with the whole model, allowing designers to evaluate the performance of the implemented subsystem under the constraints and conditions modeled within the testbench. The results of such simulations are shown in Fig. 7. A single correlator receiver is also shown just to give an idea about the performance of the other receivers. The JHDL toolset provides facilities to export EDIF netlist which allowed for mapping the implemented system in FPGA. The whole receiver was mapped in a Xilinx XCV800 FPGA using 4793 out of 9408 slices and operating at 15.672 MHz, which is more than enough for the WCDMA standard. Current research aims to address the verification of the final implementation in FPGA with the testbench implemented within PtolemyII. Such co-simulation platform will reuse the reconfigurable hardware encapsulation approach presented in [23].

5 Final Remarks

Reconfigurability plays a key role in the design of embedded systems as it helps to overcome the traditional trade-off between the performance of ASICs and the flexibility of general purpose processors. Based on a selected set of relevant applications, this paper presented various functional and architectural optimization strategies, system integration concepts and design-and-validation methodologies, proving thus the effectiveness of employing reconfigurable platforms for the development of embedded systems in terms of functionality, performance, time-to-market, life-cycle improvement, and design and maintenance costs.

References

1. Murgan, T., Petrov, M., Majer, M., Zipf, P., Glesner, M., Heinkel, U., Pleickhardt, J., Bleisteiner, B.: Adaptive architectures for an OTN processor: Reducing design costs through reconfigurability and multiprocessing. In: ACM Computing Frontiers Conf. (2004) 408–414
2. Petrov, M., Obeid, A.M., Murgan, T., Zipf, P., Brakensiek, J., Oelkrug, B., Glesner, M.: An adaptive trace-back solution for state-parallel Viterbi decoders. In: IFIP Intl. Conf. VLSI, Darmstadt, Germany (2003) 167–172

3. Petrov, M., Murgan, T., Obeid, A.M., Chiţu, C., Zipf, P., Brakensiek, J., Glesner, M.: Dynamic power optimization of the trace-back process for the Viterbi algorithm. In: IEEE Intl. Symp. Circuits and Syst., Vancouver, Canada (2004) 721–724
4. Obeid, A.M., García, A., Glesner, M.: A constraint length and throughput parameterizable architecture for Viterbi decoders. In: IEEE Int. Conf. Microelectronics. (2004)
5. Obeid, A.M., García, A., Petrov, M., Glesner, M.: A multi-path high speed Viterbi decoder. In: IEEE Int. Conf. Electronics, Circuits and Syst. (2003)
6. He, S., Torkelson, M.: Design and implementation of a 1024-point pipeline FFT processor. In: IEEE Custom Integrated Circuits Conf. (1998) 131 –134
7. Pionteck, T., Stiefmeier, T., Staake, T., Kabulepa, L., Glesner, M.: Integration dynamisch rekonfigurierbarer funktionseinheiten in prozessoren. In: Int. Conf. Architecture of Computing Systems. (2004)
8. Hinkelmann, H., Pionteck, T., Kleine, O., Glesner, M.: Prozessorintegration und speicheranbindung dynamisch rekonfigurierbarer funktionseinheiten. In: Int. Conf. Architecture of Computing Systems. (2004)
9. Rijpkema, E., Goossens, K., Rădulescu, A., Dielissen, J., van Meerbergen, J., Wielage, P., Wateralnder, E.: Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip. IEE Proc. Comput. Digital Techniques **150** (2003) 294–302
10. Kumar, S., Jantsch, A., Soininen, J.P., Forsell, M., Millberg, M., Öberg, J., Tiensyrjä, K., Hemani, A.: A network on chip architecture and design methodology. In: Proc. VLSI Ann. Symp. (2002) 105–112
11. Hollstein, T., Ludewig, R., Mager, C., Zipf, P., Glesner, M.: A hierarchical generic approach for on-chip communication, testing and debugging of SoCs. In: Proc. of VLSI-SoC. (2003)
12. Hollstein, T., Zimmer, H., Hohenstern, S., Glesner, M.: HiNoC: A flexible multi-mode transmission network-on-chip platform. In: Proc. Baltic Electronics Conf. (2004)
13. Chelcea, T., Nowick, S.M.: Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. In: Proc. Design Automation Conf., Las Vegas, CA (2001)
14. Mu, F., Svensson, C.: A 750 Mb/s 0.6 μm CMOS two-phase input port using self-tested self-synchronization. In: IEEE Int. Conf. Solid-State Circuits. (1999)
15. Mead, C., Conway, L.: Introduction to VLSI Systems. Addison-Wesley (1980)
16. Zipf, P., Hinkelmann, H., Ashraf, A., Glesner, M.: A switch architecture and signal synchronization for GALS system-on-chips. In: Proc. Symp. Integrated Circuits and Syst. Design, Porto de Galinhas, Pernambuco, Brazil (2004) 210–215
17. Xilinx Inc. http://toolbox.xilinx.com/docsan/xilinx6/books/docs/dev/dev.pdf: Development System Reference Guide. (2003)
18. Lee, E.A.., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems, and Computers **12** (2003) 231–260
19. Hooli, K.: Equalization in WCDMA Terminals. PhD thesis, Oulu University Press (2003)
20. Harada, H., Prasad, R.: Simulation and Software Radio for Mobile Communication. Artech House Publishers (2002)
21. Bellows, P., Hutchings, B.L.: JHDL - an HDL for reconfigurable systems. In: Proc. FCCM98, Napa Valley, CA (1998)
22. Wirthlin, M.: Integration of JHDL and PtolemyII. Available at http://www.ee.byu.edu/faculty/wirthlin/projects/ptjhdl.htm (2005)
23. Indrusiak, L.S., Lubitz, F., Reis, R., Glesner, M.: Ubiquitous access to reconfigurable hardware: Application scenarios and implementation issues. In: Proc. DATE. (2003) 940–945

Reconfigurable Multiple Operation Array

Humberto Calderon and Stamatis Vassiliadis

Computer Engineering Laboratory,
Electrical Engineering Dept., EEMCS, TU Delft, The Netherlands
{H.Calderon, S.Vassiliadis}@ewi.tudelft.nl
http://ce.et.tudelft.nl

Abstract. In this paper, we investigate the collapsing of eight multi-operand addition related operations into a single and common (3:2) counter array. We consider for this unit multiplication in integer and fractional representations, the Sum of Absolute Differences (SAD) in unsigned, signed magnitude and two's complement notation. Furthermore, the unit also incorporates a Multiply-Accumulation unit (MAC) for two's complement notation. The proposed multiple operation unit was constructed around 10 element arrays that can be reduced using well known counter techniques, which are feed with the necessary data to perform the proposed eight operations. It is estimated that 6/8 of the basic (3:2) counter array is shared by the operations. The obtained results of the presented unit indicates that is capable of processing a 4x4 SAD macro-block in 36.35 ns and takes 30.43 ns to process the rest of the operations using a VIRTEX II PRO xc2vp100-7ff1696 FPGA device.

1 Introduction: The Need for Reconfigurability

The need of multimedia Instruction Set Architectures (ISA) extensions with high performance processing and flexibility characteristics are potentially met with the use of reconfigurable technologies[1]. The new emerging capabilities in Reconfigurable Computing (RC) are letting us to dynamically reconfigure a portion of a FPGA. Reconfigurable fabrics can be used to support a common and basic logic blocks intended to be used in several operand addition related operations. The common blocks can be configured in advance; therefore, the hardware differences needed for performing a particular operation will be reconfigured partially based on the hardware differences between the common basic array and the new needed functionalities, instead of programming totally the new entire desired operation [2]. This work presents the collapsing of eight multi-operand addition related operations into the common hardware suitable to be implemented into a VLSI as a run time configurable unit and also over a reconfigurable technology as a reconfigurable run time unit. The multiple operation array has the following embedded units and features:

- A 16 x 16 bit multiplier for integer and fractional representations with universal notations [1].
- A 4 x 4 picture elements concurrent SAD macro-block in universal notation.

[1] Universal notation, in the context of this article, assumes operands and results to be in unsigned, sign magnitude and two's complement notations.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 22–31, 2005.

- The Multiply-Accumulation Unit (MAC) for two's complement notation.
- A performance of 35.6 ns for 4x4 SAD macro-block and 30.43 ns for the rest of 7 operations.

The paper is organized as follows. Section 2 outlines the Reconfigurable Multiple Operation Array organization. Section 3 presents the experimental results of the mapped unit, as well as other comparison units in terms of area used and time delay. Finally, the article is concluded in section 4.

2 Reconfigurable Multiple Operation Array

This section begins presenting a background and relevant work; consequently, a general array description of the Reconfigurable Multiple Operation Array is described. Finally, a complete description of the equations set for the construction and reproduction of the proposed unit are shown.

2.1 Background and Related Work

Motion estimation techniques divide an image frame for its processing in macro-blocks of $n*n$ picture elements (pels). The processing establishes if there is a difference between two image blocks using the Sum of Absolute Differences (SAD) operation, establishing the pels differences between two chosen frames. Equation (1) states the metric to evaluate the searched block.

$$\sum_{j=1}^{16}\sum_{i=1}^{16}|IN1(x+i,y+j)-IN2((x+r)+i,(y+s)+j)| \tag{1}$$

where, the duple (x,y) represents the position of the current block, and the pair (r,s) denotes the displacement of $IN2$, relative to reference block $IN1$. Different investigations including a previous author's work have been proposed to speed up the critical SAD kernel [3],[4],[5]. The processing requires that SAD input terms received by the multiple operation array have to be ordered previously, in order to compute correctly the operation; therefore, the absolute operation $\mid IN1-IN2 \mid$ can be substituted, with $IN1-IN2$ or $IN2-IN1$ depending whether $IN1$ or $IN2$ is the smallest and thus obtaining a positive result. As is suggested in [5] we can make this operation inverting one of the operands and computing the carry out of the addition of both operands as stated by the following equations:

$$\overline{IN1}+IN2 \geq 2^i-1 \tag{2}$$

therefore

$$IN2 > IN1 \tag{3}$$

means checking whether the addition of the bit inverted *IN1* and the operand *IN2* produces a carry out. The outcome determines which one is the smallest, depending on the existence or not of the carry output. Consequently, a simple logic can be used to correct the terms and feed the array. Regarding universal units capable to work with universal notations, the reader is referred to view the predecessor unit in [6] and a recent reintroduction in [7]. An extra row of (3:2)counter for MAC operation is used into the Reconfigurable Multiple Operation Array as a technique for accelerating the processing, a detailed description of this kind of approach units can be seen in [8].

2.2 The Array Description

The proposed unit has been constructed around a rectangular array that can be reduced using (3:2) counters; all ten operational fields of the array presented in Fig. 1(a), called sections in the paper, receive the integer numbers $X(i)$ and $Y(i)$, the fractional numbers $A(i)$ and $B(i)$, (all represented with 16 bits), the $W(i)$ summand represented with 32 bits for MAC operation as well as the 32 SAD terms depicted in (4) for the computation of a 4x4 macro-block:

$$I_{(j,i)} = I_{(j,15)} I_{(j,14)} \cdots \cdot I_{(j,1)} I_{(j,0)} \quad \forall \;\; 1 \leq j \leq 32 \qquad (4)$$

where the index j states the 32 inputs; and i is used to denote the positional weight of the data bits in each input. The multiplication related operations of the proposed unit requires the partial product creation as stated by the following equations: $\forall \;\; 0 \leq j \leq 15$ while $\forall \;\; 0 \leq i \leq 15$

$$Z_{(j,i)} = X_{(i)} \cdot Y_{(j)} \qquad (5)$$

$$F_{(j,i)} = A_{(i)} \cdot B_{(j)} \qquad (6)$$

All these data, feeds the (3:2) counters through 3 inputs $I1(j)(i)$, $I2(j)(i)$ and $I3(j)(i)$; and produces two outputs $S(j)(i)$ and $C(j)(i)$. The basic layout topology of the (3:2) counter is presented on Fig. 1(b) and this detail is replicated in all sections of the array; nevertheless, in the limit of both sides, left and right, the carry input $I3(j)(i)$ uses an additional multiplexor which is in charge of propagating or inhibiting the carry out of the (3:2) counters of the right side, and introduces a Hot One (HO) or a zero for SAD processing. The multiplexor presented in the aforementioned Fig. 1(b), represented by the bar, has a signal e to control the data related operations and reconfiguring in this way the operation being computed by the array. Furthermore has to be noticed that the first row of both sides, the left and the right uses three multiplexors instead of one as is described further (see equations, section 2.3).

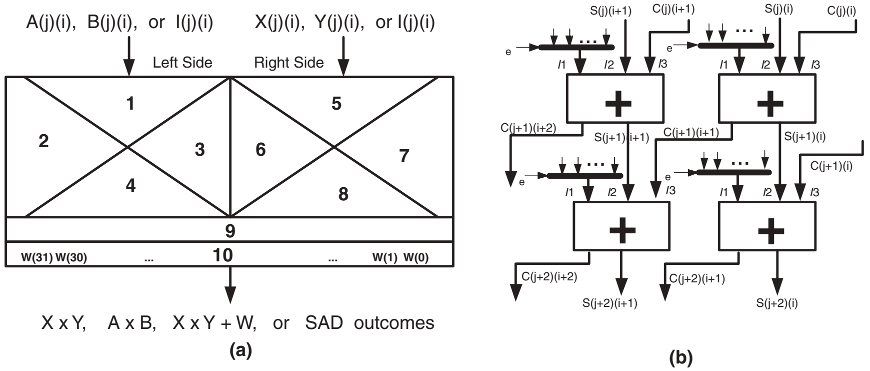


Fig. 1. The Reconfigurable Multiple Operation Array Scheme

Table 1. Sections of the Reconfigurable Multiple Operation Array

	SAD	Unsigned	Signed	Two's	MAC	Unsigned	Signed	Two's
		Integer	Integer	Integer	Two's	Fractional	Fractional	Fractional
Section 1	$I_{(j,i)}$	0	0	$Z_{(j,15)}$	$Z_{(j,15)}$	$F_{(j,i)}$	$F_{(j,i)}$	$F_{(j,i)}$
Section 2	$I_{(j,i)}$	0	0	$Z_{(j,15)}$	$Z_{(j,15)}$	0	0	$F_{(j,15)}$
Section 3	$I_{(j,i)}$	$Z_{(j,i)}$	$Z_{(j,i)}$	$Z_{(j,i)}$	$Z_{(j,i)}$	$F_{(j,i)}$	$F_{(j,i)}$	$F_{(j,i)}$
Section 4	$I_{(j,i)}$	0	0	$Z_{(j,15)}$	$Z_{(j,15)}$	0	0	$F_{(j,15)}$
Section 5	$I_{(j,i)}$	$Z_{(j,i)}$	$Z_{(j,i)}$	$Z_{(j,i)}$	$Z_{(j,i)}$	0	0	0
Section 6	$I_{(j,i)}$	$Z_{(j,i)}$	$Z_{(j,i)}$	$Z_{(j,i)}$	$Z_{(j,i)}$	$F_{(j,i)}$	$F_{(j,i)}$	$F_{(j,i)}$
Section 7	$I_{(j,i)}$	0	0	0	0	0	0	0
Section 8	$I_{(j,i)}$	0	0	0	0	$F_{(j,i)}$	$F_{(j,i)}$	$F_{(j,i)}$
Section 9	0	$Z_{(15,i)}$	$Z_{(15,i)}$	$Z_{(15,i)}$	$Z_{(15,i)}$	0	0	0
Section 10	0	0	0	0	$W_{(i)}$	0	0	0

Regarding the use of these sections, depicted in Fig. 1(a), (3:2) counters of sections 5, 6, 3 and 4 process the integer partial products described by (5). Fractional processing is achieved with sections 1, 3, 6 and 8; furthermore, sections 1 and 2 as well as 2 and 4 process the sign extensions, and zeros given the universal characteristic to the multiplier processing for integers and fractional numbers respectively. Concerning SAD processing, two main sections of the array are been used, the left and right side. The left side utilizes the sections 1, 2, 3 and 4 and the right side uses section 5, 6, 7, 8. Additionally, section 9 as part of the multiplier array, is used to add the last partial products; and section 10 receives the W(i) addend for MAC processing. Table 1 summarizes the terms received by $I1$ in each one of the main sections through the 8 to 1 multiplexor. From the table is evident that we process numbers of two complement representation with sign extension, and signed magnitude numbers are processed like positive numbers, making the sign bit zero and updating the result with the XOR of multiplicand and multiplier signs.

2.3 The Array Construction Equations Description

The multiple operation array is conformed by 32 columns and 16 rows of (3:2) counters, giving a total of 512 (3:2)counters, see Fig. 2. It can be mentioned that Fig. 2 contains all the details from Fig. 1(a). The notation used for representing (3:2) counters gives us information of the different kind of data received by these core logic blocks. The first columns of sections 3 and 7 (columns 0 and 16, see Fig. 1(a) and 2) from row 1 to 8, are used to introduce the Hot One (HO), the rest of the column introduces zero, this data is necessary for SAD calculation due to $A - B = A + \overline{B} + 1$, also column 16 inhibits the carry propagation from the right side. It should also be noted that the last row of this figure, symbolized with a plus sign $+$ represents the final adder used to calculate outcome values of the (3:2) counter array. The remainder of the section describes in a detailed way the equations for the 10 element arrays of (3:2) counter subsections, detailing for all cases the input $I1$ and also the other equations for inputs $I2$ and $I3$ when the application is different to the functionality presented in Fig. 1(b).

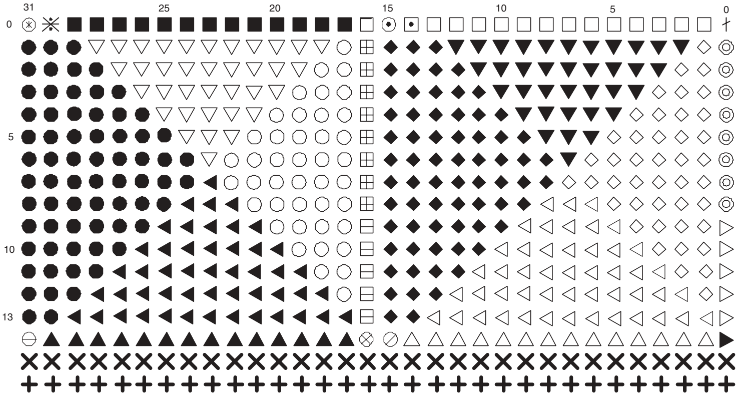


Fig. 2. The Reconfigurable Multiple Operation Array Organization

Section 1. The section is divided into four subsections; the first three are embedded into the first row.

□ : $$I1_{(0,16)} = I_{(1,0)} \cdot e_0 + Z_{(1,15)} \cdot e_1 + Z_{(1,15)} \cdot e_2 + Z_{(1,15)} \cdot e_3 + Z_{(1,15)} \cdot e_4 + F_{(15,1)} \cdot e_5 + F_{(15,1)} \cdot e_6 + F_{(15,1)} \cdot e_7$$

$$I2_{(0,16)} = I_{(2,0)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + Z_{(0,15)} \cdot e_3 + Z_{(0,15)} \cdot e_4 + F_{(14,2)} \cdot e_5 + F_{(14,2)} \cdot e_6 + F_{(14,2)} \cdot e_7$$
$$I3_{(0,16)} = I_{(3,0)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$$

■ : $\forall \quad 1 \leq i \leq \; 13$

$$I1_{(0,i+16)} = I_{(1,i)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + Z_{(0,15)} \cdot e_3 + Z_{(0,15)} \cdot e_4 + F_{(15,i+1)} \cdot e_5 + F_{(15,i+1)} \cdot e_6 + F_{(15,i+1)} \cdot e_7$$

$$I2_{(0,i+16)} = I_{(2,i)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + Z_{(1,15)} \cdot e_3 + Z_{(1,15)} \cdot e_4 + F_{(14,i+2)} \cdot e_5 + F_{(14,i+2)} \cdot e_6 + F_{(14,i+2)} \cdot e_7$$

$$I3_{(0,i+16)} = I_{(3,i)} e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$$

⋇ : $$I1_{(0,30)} = I_{(1,14)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + Z_{(1,15)} \cdot e_3 + Z_{(1,15)} \cdot e_4 + F_{(15,1)} \cdot e_5 + F_{(15,1)} \cdot e_6 + F_{(15,1)} \cdot e_7$$

$$I2_{(0,30)} = I_{(2,14)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + Z_{(0,15)} \cdot e_3 + Z_{(0,15)} \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + F_{(14,15)} \cdot e_7$$
$$I3_{(0,30)} = I_{(3,14)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$$

⊛ : $$I1_{(0,31)} = I_{(1,15)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + Z_{(1,15)} \cdot e_3 + Z_{(1,15)} \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + F_{(15,1)} \cdot e_7$$

$$I2_{(0,31)} = I_{(2,15)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + Z_{(0,15)} \cdot e_3 + Z_{(0,15)} \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + F_{(14,15)} \cdot e_7$$
$$I3_{(0,31)} = I_{(3,15)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$$

∇ **:** Let $n = 2$, $m = 12$; $\forall\ 1 \le j \le 6$ and; $\forall\ n \le i \le m$,with $m = m - 1$; and ;$n = n + 1$; for each column

$$I1_{(j,i+16)} = I_{(j+3,i)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + Z_{(j+1,15)} \cdot e_3 + Z_{(j+1,15)} \cdot e_4 + F_{(14-j,i+3)} \cdot e_5 + F_{(14-j,i+3)} \cdot e_6 + F_{(14-j,i+3)} \cdot e_7$$

Section 2. The section has the two representative equations:

● **:** Let $n = 13$; $\forall\ 1 \le j \le 6$ and $\forall\ n \le i \le 15$, with $n = n - 1$

$$I1_{(j,i+16)} = I_{(j+3,i)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + Z_{(j+1,15)} \cdot e_3 + Z_{(j+1,15)} \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + F_{(14-j,15)} \cdot e_7$$

● **:** Let $n = 8$; $\forall\ 7 \le j \le 13$ and $\forall\ n \le i \le 15$, with $n = n + 1$

$$I1_{(j,i+16)} = I_{(j+3,i)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + Z_{(j+1,15)} \cdot e_3 + Z_{(j+1,15)} \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + F_{(14-j,15)} \cdot e_7$$

Section 3. The section is subdivided into four main parts:

⊞ **:** $\forall\ 1 \le j \le 8$

$$I1_{(j,16)} = I_{(j+3,0)} \cdot e_0 + Z_{(j+1,15-j)} \cdot e_1 + Z_{(j+1,15-j)} \cdot e_2 + +Z_{(j,16-j)} \cdot e_3 + Z_{(j,16-j)} \cdot e_4 + F_{(14-j,j+2)} \cdot e_5 + F_{(14-j,j+2)} \cdot e_6 + F_{(14-j,j+2)} \cdot e_7$$

$$I3_{(j,16)} = 1 \cdot e_0 + C_{(j-1,i+15)} \cdot e_1 + C_{(j,i+16)} \cdot e_2 + C_{(j,i+16)} \cdot e_3 + C_{(j,i+16)} \cdot e_4 + C_{(j,i+16)} \cdot e_5 + C_{(j,i+16)} \cdot e_6 + C_{(j,i+16)} \cdot e_7$$

⊟ **:** $\forall\ 9 \le j \le 13$

$$I1_{(j,16)} = I_{(j+3,0)} \cdot e_0 + Z_{(j+1,15-j)} \cdot e_1 + Z_{(j+1,15-j)} \cdot e_2 + Z_{(j,15-j)} \cdot e_3 + Z_{(j,15-j)} \cdot e_4 + F_{(14-j,j+2)} \cdot e_5 + F_{(14-j,j+2)} \cdot e_6 + F_{(14-j,j+2)} \cdot e_7$$

$$I3_{(j,16)} = I3_{(j,16)} = 1 \cdot e_0 + C_{(j,i+16)} \cdot e_1 + C_{(j,i+16)} \cdot e_2 + C_{(j,i+16)} \cdot e_3 + C_{(j,i+16)} \cdot e_4 + C_{(j,i+16)} \cdot e_5 + C_{(j,i+16)} \cdot e_6 + C_{(j,i+16)} \cdot e_7$$

First part ○ **:** Let $m = 1$; $\forall\ 1 \le j \le 6$ and $\forall\ 1 \le i \le m$, with $m = m + 1$

$$I1_{(j,i+16)} = I_{(j+3,i)} \cdot e_0 + Z_{(j+1,i+14)} \cdot e_1 + Z_{(j+1,i+14)} \cdot e_2 + Z_{(j+1,i+14)} \cdot e_3 + Z_{(j+1,i+14)} \cdot e_4 + F_{(14-j,i+3)} \cdot e_5 + F_{(14-j,i+3)} \cdot e_6 + F_{(14-j,i+3)} \cdot e_7$$

Second part ○ **:** Let $m = 6$; $\forall\ 7 \le j \le 12$ and $\forall\ 1 \le i \le m$, with $m = m - 1$

$$I1_{(j,i+16)} = I_{(j+3,i)} \cdot e_0 + Z_{(j+1,i+15-j)} \cdot e_1 + Z_{(j+1,i+15-j)} \cdot e_2 + Z_{(j+1,i+15-j)} \cdot e_3 + Z_{(j+1,i+15-j)} \cdot e_4 + F_{(14-j,i+j+2)} \cdot e_5 + F_{(14-j,i+j+2)} \cdot e_6 + F_{(14-j,i+j+2)} \cdot e_7$$

Section 4. ◀ **:** Let $n = 7$, $m = 7$; $\forall\ 7 \le j \le 13$ and $\forall\ n \le i \le m$; with $n = n - 1$ and $m = m + 1$

$$IN1_{(j,i+16)} = I_{(j+3,i)} \cdot e_0 + Z_{(j+1,15-j+i)} \cdot e_1 + Z_{(j+1,15-j+i)} \cdot e_2 + Z_{(j+1,15-j+i)} \cdot e_3 + Z_{(j+16,15-j+i)} \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + F_{(14-j,15)} \cdot e_7$$

Section 5. It is divided into four subsections; the first three are embedded in the first row.
†: $I1_{(0,0)} = I_{(17,0)} \cdot e_0 + Z_{(0,0)} \cdot e_1 + Z_{(0,0)} \cdot e_2 + Z_{(0,0)} \cdot e_3 + Z_{(0,0)} \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$
$I2_{(0,0)} = I_{(18,0)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$
$I3_{(0,0)} = I_{(19,0)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$

□: $\forall \ 1 \leq i \leq \ 13:$
$I1_{(0,i)} = I_{(17,i)} \cdot e_0 + Z_{(0,i)} \cdot e_1 + Z_{(0,i)} \cdot e_2 + Z_{(0,i)} \cdot e_3 + Z_{(0,i)} \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$
$I2_{(0,i)} = I_{(18,i)} \cdot e_0 + Z_{(1,i-1)} \cdot e_1 + Z_{(1,i-1)} \cdot e_2 + Z_{(1,i-1)} \cdot e_3 + Z_{(1,i-1)} \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$
$I3_{(0,i)} = I_{(19,i)} e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$

⊡: $I1_{(0,14)} = I_{(17,14)} \cdot e_0 + Z_{(0,14)} \cdot e_1 + Z_{(0,14)} \cdot e_2 + Z_{(0,14)} \cdot e_3 + Z_{(0,14)} \cdot e_4 + F_{(14,0)} \cdot e_5 +$
$F_{(14,0)} \cdot e_6 + F_{(14,0)} \cdot e_7$

$I2_{(0,14)} = I_{(18,14)} \cdot e_0 + Z_{(1,13)} \cdot e_1 + Z_{(1,13)} \cdot e_2 + Z_{(1,13)} \cdot e_3 + Z_{(1,13)} \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$
$I3_{(0,14)} = I_{(19,14)} \cdot e_0 + 0 e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$

⊙: $I1_{(0,15)} = I_{(17,15)} \cdot e_0 + Z_{(0,15)} \cdot e_1 + Z_{(0,15)} \cdot e_2 + Z_{(0,15)} \cdot e_3 + Z_{(0,15)} \cdot e_4 + F_{(15,0)} \cdot e_5 +$
$F_{(15,0)} \cdot e_6 + F_{(15,0)} \cdot e_7$
$I2_{(0,15)} = I_{(18,15)} \cdot e_0 + Z_{(1,14)} \cdot e_1 + Z_{(1,14)} \cdot e_2 + Z_{(1,14)} \cdot e_3 + Z_{(1,14)} \cdot e_4 + F_{(14,1)} \cdot e_5 +$
$F_{(14,1)} \cdot e_6 + F_{(14,1)} \cdot e_7$
$I3_{(0,15)} = I_{(19,15)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + HO \cdot e_3 + HO \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + HO \cdot e_7$

▽: Let $n = 2$, $m = 12$; $\forall \ 1 \leq j \leq \ 6$ and $\forall \ n \leq i \leq \ m$,with $m = m - 1$; and $n = n + 1$;

$I1_{(j,i+16)} = I_{(j+19,i)} \cdot e_0 + Z_{(j+1,i-j-1)} \cdot e_1 + Z_{(j+1,i-j-1)} \cdot e_2 Z_{(j+1,i-j-1)} \cdot e_3 + Z_{(j+1,i-j-1)} \cdot e_4 +$
$0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$

Section 6. The section has the two representative equations:
•: Let $n = 13$; $\forall \ 1 \leq j \leq \ 6$ and $\forall \ n \leq i \leq \ 15$, with $n = n - 1$

$$I1_{(j,i)} = I_{(j+19,i)} \cdot e_0 + Z_{(j+1,i-j-1)} \cdot e_1 + Z_{(j+1,i-j-1)} \cdot e_2 + Z_{(j+1,i-j-1)} \cdot e_3 + Z_{(j+1,i-j-1)} \cdot e_4 + F_{(14-j,i-14+j)} \cdot e_5 + F_{(14-j,i-14+j)} \cdot e_6 + F_{(14-j,i-14+j)} \cdot e_7$$

•: Let $n = 8$; $\forall \ 7 \leq j \leq \ 13$ and $\forall \ n \leq i \leq \ 15$, with $n = n - 1$

$$I1_{(j,i)} = I_{(j+19,i)} \cdot e_0 + Z_{(j+1,i+j-1)} \cdot e_1 + Z_{(j+1,i+j-1)} \cdot e_2 + Z_{(j+1,i+j-1)} \cdot e_3 + Z_{(j+1,i+j-1)} \cdot e_4 + F_{(14-j,i-14+j)} \cdot e_5 + F_{(14-j,i-14+j)} \cdot e_6 + F_{(14-j,i-14+j)} \cdot e_7$$

Section 7. The section is conformed by three subsections:
◎: $\forall \ 1 \leq j \leq \ 8,$
$I1_{(j,0)} = I_{(j+19,i)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$
$I3_{(j,0)} = 1 \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$

▷ : Let $n = 1$, $m = 6$, $\forall \ 9 \leq j \leq \ 13$, with $m = m - 1$
$I1_{(j,0)} = I_{(j+19,i)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$; $I3_{(0,i)} = 0$
◊ : Let $m = 1$; $\forall \ 1 \leq j \leq \ 7$ and $\forall \ 1 \leq i \leq \ m$, with $m = m + 1$

$I1_{(j,i+16)} = I_{(j+19,i)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$

$\Diamond$ **:** Let $m = 6$; $\forall \ \ 8 \leq j \leq 12$ and $\forall \ \ 1 \leq i \leq m$, with $m = m - 1$

$I1_{(j,i)} = I_{(j+19,i)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$

Section 8. $\lhd$ **:** Let $n = 7$, $m = 8$; $\forall \ \ 9 \leq j \leq 13$ and $\forall \ \ n \leq i \leq m$, with $n = n - 1$ while $m = m + 1$

$$I1_{(j,i)} = I_{(j+17,i)} \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + F_{(16-j,16-j-i)} \cdot e_5 + F_{(16-j,16-j-i)} \cdot e_6 + F_{(16-j,16-j-i)} \cdot e_7$$

Section 9. The section is divided into six subsections.

$\blacktriangleright$ **:** $I1_{(14,0)} = 0 \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + F_{(0,0)} \cdot e_5 + F_{(0,0)} \cdot e_6 + F_{(0,0)} \cdot e_7$

$\vartriangle$ **:** $\forall \ \ 1 \leq i \leq 14$,

$$I1_{(14,i)} = 0 \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + 0 \cdot e_4 + F_{(0,i)} \cdot e_5 + F_{(0,i)} \cdot e_6 + F_{(0,i)} \cdot e_7$$

$\oslash$ **:** $I1_{(14,15)} = 0 \cdot e_0 + Z_{(15,0)} \cdot e_1 + Z_{(15,0)} \cdot e_2 + Z_{(15,0)} \cdot e_3 + Z_{(15,0)} \cdot e_4 + F_{(0,15)} \cdot e_5 + F_{(0,15)} \cdot e_6 + F_{(0,15)} \cdot e_7$

$\otimes$ **:** $I1_{(14,16)} = 0 \cdot e_0 + Z_{(15,1)} \cdot e_1 + Z_{(15,1)} \cdot e_2 + Z_{(15,1)} \cdot e_3 + Z_{(15,1)} \cdot e_4 + F_{(0,15)} \cdot e_5 + F_{(0,15)} \cdot e_6 + F_{(0,15)} \cdot e_7$

$\blacktriangle$ **:** $\forall \ \ 1 \leq i \leq 14$

$$I1_{(14,i+16)} = 0 \cdot e_0 + Z_{(15,i+1)} \cdot e_1 + Z_{(15,i+1)} \cdot e_2 + Z_{(15,i+1)} \cdot e_3 + Z_{(15,i+1)} \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + F_{(0,15)} \cdot e_7$$

$\ominus$ **:** $I1_{(14,31)} = 0 \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + Z_{(15,15)} \cdot e_3 + Z_{(15,15)} \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + F_{(0,15)} \cdot e_7$

Section 10. $\forall \ \ 0 \leq i \leq 31$; with $n = n - 1$ while $m = m + 1$

$I1_{(j,i)} = 0 \cdot e_0 + 0 \cdot e_1 + 0 \cdot e_2 + 0 \cdot e_3 + W(i) \cdot e_4 + 0 \cdot e_5 + 0 \cdot e_6 + 0 \cdot e_7$

3 Experimental Results and Analysis

The Reconfigurable Multiple Operation Array with all necessary control logic and the Carry Unit were described using VHDL, synthesized and proved with ISE 5.2i Xilinx environment [9], for the VIRTEX II PRO xc2vp100-7ff1696 FPGA device. Additionally, all operations such us: unsigned multiplier, signed magnitude multiplier, two's complement multiplier, rectangular SAD unit (half of the rectangular array), and MAC unit were implemented individually without the overhead logic. Furthermore, previous presented units like the Universal SAD-Multiplier array (U-SAD-M) [7] for integer numbers and Baugh and Wooley (B&W) signed multiplier [10] were synthesized with the same tool in order to have more comparison parameters with our new proposal. Table 2 summarizes the performance in terms of time delay of those structures.

The additional logic introduced into the core array reduces the performance of the functionalities, as can be seen in table 2. The extra delay in terms of time is between 12 % for a 16 bits MAC operand in the proposed unit, and up to 50 % for a 32 bits MAC over the previous single functionality units like Baugh-Wooley or a simple integer U-SAD-M unit. This extra time delay diminishes to 27 % when the fast carry logic provided into the FPGAs Xilinx is used [11] in the final adder. The additional extra delays of the proposed unit compared with the previous ones are due two principal factors: the first one is related to the multiplexor used to feed the data into the input $I1$, which presents a constant delay for all the logic blocks, and the extra multiplexor introduced in the array to separate logically the right and left sides, given the possibility of propagating or inhibiting the carry and also force a Hot One or a zero for SAD processing. An acceleration into the processing can be achieved using a parallel reduction tree. Instead of using a regular array, a Wallace tree [12] can be implemented in order to accelerate the performance of the operations. Considering that $n(h) = \llcorner 3n(h-1)/2 \lrcorner$ quantifies the number of necessary levels h of (3:2) counters for the reduction of n input bits, we estimate that a logic delay has been reduced in 3.22 ns and cut 4.65 ns off in routing delay, accelerating the processing in 7.87ns. This amount of time represents an improvement of a 25.87 % in the total array delay.

Table 2. Multiple Operation Array Unit and related units (RCA final adder: ‡ : LUT based ; $*$: Based on Xilinx Fast Carry Logic [11])

	Time delay			Hardware use		
Unit	Logic Delay (ns)	Wire Delay (ns)	Total Delay (ns)	Slices	LUTs	IOBs
SAD ‡	13.184	12.006	25.191	242	421	272
Unsigned Multiplier ‡	14.589	14.639	29.282	300	524	64
Two's I ‡	12.595	15.564	28.159	294	511	64
Two's F ‡	15.153	16.825	31.978	443	770	64
Baugh&Wooley ‡	15.555	15.826	31.381	330	574	65
U-SAD-M ‡	15.877	15.741	31.618	686	1198	322
U-SAD-M $*$	14.311	9.568	23.879	658	1170	322
MAC ‡	15.062	19.064	34.662	358	622	96
Our Proposal ‡	21.351	26.040	47.391	1373	2492	643
Our Proposal-16 MAC ‡	16.521	19.065	35.586	1360	2465	643
Our Proposal $*$	15.311	15.127	30.438	1354	2458	643
Carry Unit	2.576	3.338	5.914	35	61	64

Concerning the silicon used by the Reconfigurable Multiple Operation Unit and the other structures, depicted on Table 2, the overhead in terms of hardware of the presented unit is considerable and is the paid price for its multi-functional characteristic. Nevertheless, if the eight units are implemented separately we will need two times the hardware resources and we will have one third of additional bandwidth needs in the worst case scenario. We should also consider that 6/8 of the basic logic block array are shared by the eight operations making this chosen operations a good candidates for it's implementation with partial reconfiguration paradigm, based on differences of the functional units.

4 Conclusions

We have presented a novel Reconfigurable Multiple Operation Array organization. The proposed unit can be implemented on a VLSI intended to be used as a run time configurable unit, and it can also be used in a reconfigurable technology as a run time reconfigurable unit. The whole array is configured using multiplexors, which can be replaced with faster connections on a partially reconfigurable environment. Several units are been coded and synthesized to have a wide comparison environment, furthermore, a brief analysis of the obtained results in terms of area used and time delay are presented given a maximum work frequency of 27.5 MHz for the calculus for a 4x4 SAD macroblock and a 32.85 MHz for MAC and the other multiplier operations in a VIRTEX II PRO device using a 3% of the available slices of the chosen FPGA.

References

1. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Panainte, E.: The molen polymorphic processor. IEEE Transactions on Computers (2004) 1363 – 1375
2. Xilinx: Two flows for partial reconfiguration: Module based or difference based. Application Note:Virtex, Virtex-E, Virtex-II, Virtex II Pro Families,XAPP290 (2003) 1 – 28
3. Guevorkian, D., Launiainen, A., Liuha, P., Lappalainen, V.: Architectures for the sum of absolute differences operation. IEEE Workshop on Signal Processing Systems (SPIS'02) (2002)
4. Kuhn, P.: Fast mpeg-4 motion estimation: Processor based and flexible vlsi implementations. Journal of VLSI Signal Processing (1999) 67–92
5. Vassiliadis, S., Hakkennes, E., Wong, S., Pechanek., G.: The sum-absolute-difference motion estimation accelerator. Proceedings of Euromicro Conference, 24th (1998) 559–566
6. Vassiliadis, S., Schwarz, E., Putrino, M.: Quasi-universal vlsi multiplier with signed digit arithmetic. Proceedings of the 1987 IEEE,Southern Tier Technical Conference (1987) 1–10
7. Calderon, H., Vassilidis, S.: Reconfigurable universal sad-multiplier array. Accepted for publication in: Proccedings of ACM international conference - Computer Frontiers (2005)
8. Yaday, N., Schulte, M., Glossner, J.: Parallel saturating fractional arithmetic units. Procceedings of the Ninth great lakes Symposium on VLSI (1999)
9. Xilinx: The xilinx software manuals, xilinx 5.2i. http://www.xilinx.com/support/sw_manuals/xilinx5/index.htm (2003)
10. Baugh, C., Wooley, B.: A two's complement parallel array multiplication algorithm. IEEE, Transactions on Computers (1973) 1045–1047
11. Xilinx: Virtex ii pro platform fpga handbook. (2002)
12. Wallace, C.S.: A suggestion for a fast multiplier. IEEE, Transactions on Electronic Computers (1964) 14 – 17

RAPANUI: Rapid Prototyping for Media Processor Architecture Exploration

Guillermo Payá Vayá, Javier Martín Langerwerf, and Peter Pirsch

Institute of Microelectronic Systems, University of Hannover,
Appelstr.4, 30167 Hannover, Germany
{guipava, jamarlan, pirsch}@ims.uni-hannover.de

Abstract. This paper describes a new rapid prototyping-based design framework for exploring and validating complex multiprocessor architectures for multimedia applications. The new methodology combines a typical ASIC flow with an FPGA flow focused on rapid prototyping. In order to make an exhaustive verification of the system architecture, a reference model that specifies the hardware implementation is used for validating both, HDL description and emulated system. Functional coverage in addition to traditional code coverage is used to test 100% of data, control and structural hazards of the system architecture. The reference model is also part of a stand-alone simulation environment. This allows hardware and application development be supported by a unique system model.

1 Introduction

Nowadays, emerging high-density field-programmable gate arrays (FPGA) allow to prototype complex VLSI designs (with approximately million system gates). This tremendous progress in FPGA technology provides the possibility of integrating *rapid prototyping* into a new methodology for exploring and evaluating multimedia processor systems. The use of a prototyping based framework has several advantages:

- *Architecture exploration and evaluation.* Software architecture simulators cannot provide enough information about the impact of new architectural modifications on the final full-scale implementation. Otherwise, developing a prototype easily resolves questions about design complexity and technical feasibility [1, 2]. By using FPGA emulation systems, VLSI designers can test their preliminary ideas obtaining immediate hardware metrics and studying their feasibility. Moreover, an interactive design approach can be done until the expected specifications are reached.
- *Hardware architecture simulator.* Because software simulation takes usually much too long, hours or days for simulating a few minutes of a video application sequence, a flexible real-time prototyping of the full architecture is mandatory. Moreover, early in the design cycle, by using an efficient graphical environment that interacts with the emulation system, application designers could start programming real applications.
- *Architecture validation.* Conceptual problems can be detected by simulating the whole system. In case of a multiprocessor system, communications between the

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 32–40, 2005.

different processors can be validated. Moreover, measures like maximum operations per second or memory bandwidth for real world applications can be obtained.

The RAPANUI project proposes a new prototyping-based methodology for media processor architecture exploration, where in addition of testing the performance and correctness of the design, we can run a specific media application under ”real-time” conditions for reduced image resolution using the emulated hardware. The goal of our methodology is to obtain an exhaustively verified HDL description of our processor or multiprocessor system that will finally be implemented as an ASIC. Therefore, a common source is used for both, the ASIC and FPGA implementation, respectively. Only few target-specific modules will be used for an efficient FPGA synthesis.

This paper is organized as follows. Section 2 introduces some current approaches for architecture exploration. Section 3 presents the RAPANUI methodology, placing emphasis on the different steps of our design flow. After that, some comments referring to the prototyping technology used are given. Finally, conclusions are discussed in Section 4.

2 Related Work

In the last years, a large number of simulators have been used to investigate microprocessor design issues. Functional and performance simulators have been created in order to study topics ranging from low power to high performance memory communication architectures. Although in recent years simulator environments, like SimpleScalar [3], SimOS [4] or SimICS [5] can be used to describe complex architectures, a software simulator cannot provide enough information about the impact on the implementation of new architectural modifications.

Due to the increasing complexity and capacity of programmable logic devices, FPGA-based prototypes are recently used in ASIC development to enable early validation of whole system architectures. The prototyping technologies allow an acceleration of processor development. The new hardware emulation systems [6] provide prototyping of systems ranging from simple microprocessor [2, 7, 8, 9] to complex microprocessor designs [10].

Prototype technologies should be used as a part of an ASIC flow using a common source code. Developing low-level synthesis models would cost too much time and effort.

3 Methodology Overview

In this section, we describe our methodology for processor development acceleration based on a rapid prototyping framework. In the following subsections the proposed design flow is introduced in more detail. After that, the rapid prototyping system we have used is presented.

3.1 Design Flow

The proposed methodology is divided in five successive steps (see Fig. 1). In order to develop a system that works efficiently with a specific application (or a group of similar applications), it is necessary to make an architecture-independent performance analysis. The results of this analysis support architecture design decisions made in step 2. Step 3 represents the core of our methodology and is divided in two branches, both describing the architecture to be designed, but from two different points of view: an HDL description and a reference model architecture. This reference model can be seen as an executable specification and is used to perform an IP verification of our architecture and to create a stand-alone simulator for application development. Step 4 in the diagram represents a necessary step before being able to run an application on the emulated system. On the one hand, the HDL description has to be augmented with emulation target-specific primitives. On the other hand, the programming environment is used for developing an application. Moreover, semi automatically generated programs [7] to verify the emulated system assure a 100% functional coverage.

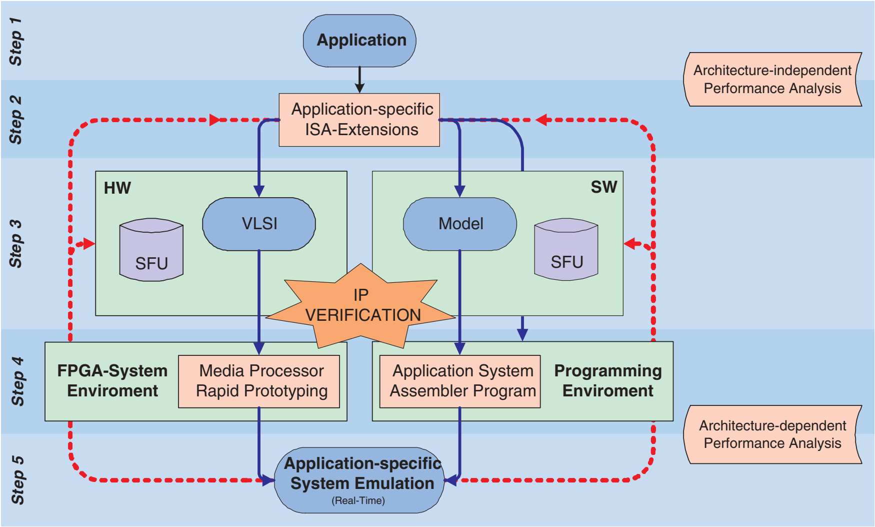

Step 1
Step 2
Step 3
Step 4
Step 5
Application
Architecture-independent Performance Analysis
Application-specific ISA-Extensions
HW
SW
VLSI
Model
SFU
SFU
IP VERIFICATION
FPGA-System Enviroment
Media Processor Rapid Prototyping
Application System Assembler Program
Programming Enviroment
Architecture-dependent Performance Analysis
Application-specific System Emulation
(Real-Time)

Fig. 1. Step diagram of the RAPANUI methodology

Our design flow includes the necessary feedback (dot lines) to eventually modify architecture parameters that improve the results obtained in the performance analysis until specifications established in Step 1 are reached.

Step 1. Application Analysis. System Level Design
In order to estimate the processing demands of a streaming media application, a platform-independent performance analysis has to be performed [11, 12]. From the analysis we derive a complexity profile which is characteristic for a specific application. These results allow us to identify optimization potential and hence will be used to guide the implementation process.

However, design issues at the system level have to be analyzed in this step. These issues include different types of parallelism at thread or process level [13] and memory bandwidth requirements [14]. They have a strong influence on latter performance analysis results. Therefore, system level design decisions have to be taken before designing the system or each processing element.

In case of implementing a multiprocessor system, the specific application has to be split up in different tasks to be spread among the different processors. Steps 2 and 3 will be independent for every single processor in order to validate their correct execution (single processor validation). Steps 4 and 5 will be performed on the complete system, which includes all verified elements.

Step 2. Application-Specific ISA-Extensions. Architecture Level Design
The architecture-independent complexity profile for a specific task in the main application gives also an important clue for defining the instruction-set architecture of the processor to be designed. In this step, different architecture alternatives of *special function units* (SFU) within the processor are discussed and analyzed [15]. What we aim at is to specify the processor on its architecture level, also considering instruction-level and data-level parallelism [16].

A complete specification of the processor to design will be done. The complete pipeline will be defined, to design later in Step 3 the verification strategy. Also, a complete list of the data, control and structural hazards and their respective solutions will be examined [17] and specified. Structural hazards might appear in a multiple issue processor [18]. All this specifications will appear in the reference model which validates, in an early phase, the functionality of the proposed processor architecture.

Step 3. Media Processor Design and Functional Verification. Register-Transfer Level Design
At this level, as we can see in Fig. 1, the design flow splits up into two branches: A register-transfer level HDL description, and the development of a software model of the architecture. This architecture reference model is written in an object-oriented programming language called OpenVera [19] and standard C. This reference model is used not only to perform an IP verification of the hardware description, but also to build a stand-alone simulator for application development. The Vera language is used because of the features it provides to facilitate testbench creation and structured RTL interfacing. Moreover, Vera simplifies the random stimulus generation and supports a functional coverage system. This system is able to monitor all states and state transitions created by the user in order to measure the ”functional” coverage [20]. This feature and the ability of verify the status of the hardware implementation via the software model, make it possible to test 100% of the processor functionality, as shown in Fig. 2. Additionally traditional code coverage should also be made.

- *HDL Processor Description.* An HDL description of the processor architecture is done, creating a library with different SFUs that can be reused in other processors (see Fig. 1). In order to reuse code, a common source for ASIC and FPGA implementations is used. The main focus is to achieve good synthesis results for ASIC implementation. Nevertheless, due to the different special resources of a specific FPGA target, synthesis efficiency is influenced significantly by matching the resources inferred by the HDL description. Therefore, some specific source code modifications for efficient FPGA synthesis have to be performed [21].

– *Architecture Reference Model.* A cycle- and bit-true architecture description is made using an object-oriented language (OpenVera) and standard C. The reference model has the same pipeline structure as the processor architecture under verification, and allows the validation of the proposed architecture in an early phase.

Verification of the HDL description is done by comparing the contents of all pipeline registers, different special registers and the whole register file for every cycle. In other to test the correct execution our processor in all conflictive possible situations, functional verification is used. Functional verification consists on creating different sample states and sample transitions, that abstract each status the processor could have and the dynamic behavior. Several counters are assigned to each sample and incremented during the simulation execution every time a state is active or a transition happened. By using semi automatically generated programs, all possible hazard situations can be checked.

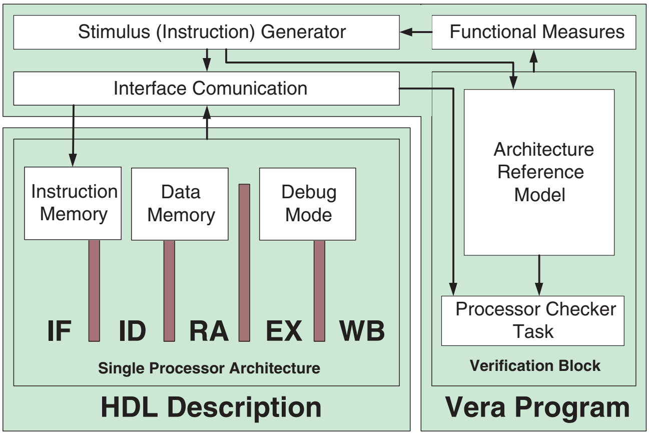


Fig. 2. Simulation scheme in Step 3

Step 4. FPGA Prototyping and Application Developing

In this Step, the HDL code of the media processor is synthesized and implemented semi-automatically on the emulation system. By using ProDesign's HDL Bridge [22], the HDL code is implemented on the CHIPit emulation Platform [6] and connects them through ProDesign's UMRBus to a simulator software (Modelsim) running on the host computer. A typical flow will consist of implementing the Design Under Test (DUT) on the CHIPit Platform and using the HDL Bridge interface and the UMRBus to verify the DUT with a ”software” testbench (see Fig. 3).

The CHIPit Platform supports the Signal Tracker hardware debugger solution from Xilinx which allows readback functionality of Virtex and Virtex-II FPGAs [23]. Thus, internal register output signals of the DUT will be visible in the simulator.

Two possible verification strategies could be used in this Step.

- *Signal Tracker.* Modelsim simulator can be used with Vera and HDL Bridge. So using the signal tracker solution, the same verification strategy as in Step 3 could be used in a new co-emulated environment.
- *Debug mode.* An interface to control the ”debug mode” of our processor is used (see Fig. 4) together with the architecture reference model from Step 3 to validate the processor. Using this debug mode, the hardware implemented on the FPGA is close to the hardware implemented on the ASIC, so the possible modifications when not using the debug mode do not affect the system validation.

In this Step, a full multiprocessor system validation can be done. Instead of using the ”debug mode”, system communication overhead can be reduced by interacting with or simulating the system controller, which controls the other processors.

Optionally, the architecture reference model can be used to run software versions of different tasks of an application. This feature offers the possibility of using the reference model not only as a verification instrument but also as a standalone simulator, which is very useful to assist application development.

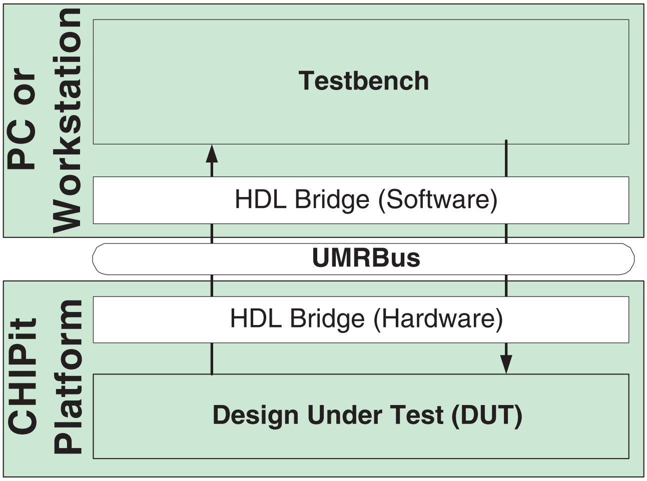


Fig. 3. Typical verification flow using the CHIPit Platform

Step 5. Application-Specific System Emulation

The FPGA-System environment is used to run different applications under ”real-time” conditions, i.e., with reduced image resolution or rescaling adequately the time execution results to a final ASIC implementation. An architecture-dependent performance analysis checks whether the expected requirements for the selected tasks are reached or not.

In case of not reaching the required performance results, several iterative improvement steps are possible. New extensions can be added in Step 2 or new architectural

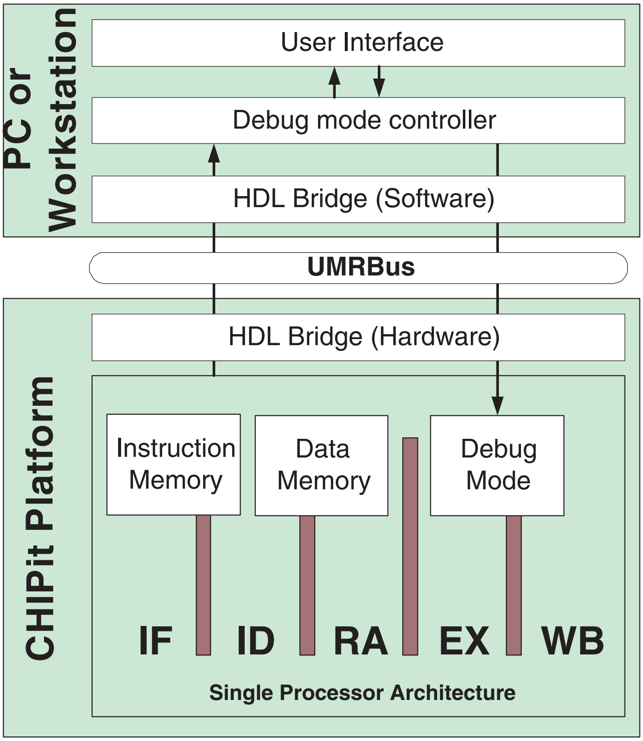


Fig. 4. Emulation scheme in Step 4 and Step 5

modifications to decrease critical paths can be done in Step 3 for any single processor. Even adding an additional processor can be done quite easy, in order to improve the overall system performance.

3.2 Prototyping Technology

The availability of high-density low-cost FPGAs has made rapid prototyping of complex multiprocessor systems possible. In only a few years, many commercial prototyping boards have appeared. External SDRAMs, I/O interfaces with external audio and video codecs, make prototyping boards suitable to emulate even multimedia systems.

Our prototype environment is based on a ProDesign emulation system. ProDesign [6] presents a new FPGA based verification system that provides enough capacity to prototype relatively complex media processor architectures. The CHIPit® Platinum Edition is the latest member of the CHIPit product family, containing a scalable number of Xilinx XC2V6000 or 8000 Virtex-II FPGAs, which allows prototyping of systems with up to 10M ASIC gates. Extensions, like memory boards, displays or interfaces, enhance the emulation performance and connectivity of the system.

Table 1. CHIPit® Platinum Edition

Specifications	FPGA System
FPGA type	Xilinx Virtex-II XC2V6000 or 8000
Approx. ASIC gates	3.3 M
FPGA system gates	48 M
Total number of block RAM	18 Mbits
Max. Speed on FPGA board	about 200 MHz
Max. total system speed	about 100 MHz
Max. number of external clocks	24
Max. number of High Speed clocks	8
Max. number of user I/Os	1920

4 Conclusions

The RAPANUI methodology is a new rapid prototyping-based design framework for media processor architecture exploration and validation. RAPANUI describes a new design flow to accelerate multimedia processor designs, generating intermediate feedbacks to eventually modify architecture parameters that improve the overall system performance.

The proposed methodology is divided into five steps which allow the designer to explore and validate multiprocessor architectures during the design phase. These steps comprise: 1. application analysis, 2. application-specific ISA-extensions, 3. media processor design and functional verifications, 4. FPGA prototyping and applications developing, and 5. application-specific system emulation.

During the design flow, four important concepts are considered: *performance analysis* to evaluate the multimedia application and the multimedia processor, *architecture reference model* to specify the processor system that can be used as a simulator as well, *functional coverage* in addition to traditional code coverage to test 100% of data, control and structural hazards and *FPGA prototyping* to validate and emulate the processor system in a hardware environment.

Finally, this new methodology augments a typical ASIC flow with an FPGA flow focused on rapid prototyping.

The next step in our project will be the validation of our methodology by verifying a full multiprocessor system. Emulation will be performed by interacting only with one single processor, in order to reduce communication overhead. This processor, called system controller, is in charge of managing the full multiprocessor system.

References

1. Ray, J., Hoe, J.: High-Level Modeling and FPGA Prototyping of Microprocessors. In: Proceedings of Int. Symposium on Field Programmable Gate Arrays. (2003) 100–107
2. Brown, R., Hayes, J., Mudge, T.: Rapid Prototyping and Evaluation of High-Performance Computers. In: Proceedings of the 1996 Conference on Experimental Research in Computer Architectures. (1996) 159–168

3. Burger, D., Austin, T.: The SimpleScalar Tool Set, Version 2.0. Technical Report 1342 (1997)
4. Rosenblum, M., Herrod, S., Witchel, E., Gupta, A.: Complete Computer System Simulation: The SimOS approach. IEEE Parallel and Distributed Technology: Systems and Applications **3** (1995) 34–43
5. Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Magnusson, P., Moestedt, A., Werner, B.: Simics: A Full System Simulation Platform. Computer **35** (2002) 50–58
6. ProDesign Electronic GmbH: (2005) http://www.prodesigncad.de/.
7. Martínez-Pérez, J., Ballester-Merelo, F., Herrero-Bosch, V., Gadea-Gironés, R.: Ariadna: An FPGA-Oriented 32-bit Processor Core Using Synopsys Flow. In: Synopsys User Group Workshop. (2003)
8. Gshwind, M., Salapura, V., Maurer, D.: FPGA Prototyping of a RISC Processor Core for Embedded Applications. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **9** (2001) 241–250
9. Kim, Y., Kim, T.: A Design and Tool Reuse Methodology for Rapid Prototyping of Application Specific Instruction Set Processors. In: Proceedings of the 1999 IEEE Workshop on Rapid System Prototyping. (1999) 46–51
10. Gateley, J., Greenley, D., Blatt, M., Chen, D., Cooke, S., Dasai, P., Doreswanny, M., Elgood, M., Feierbach, G., Goldsbury, T.: UltraSPARC-I Emulation. In: Proceedings of the 32nd ACM/IEEE Design Automation Conference. (1995) 13–18
11. Reuter, C., Martín, J., Stolberg, H.J., Pirsch, P.: Performance Estimation of Streaming Media Applications for Reconfigurable Platforms. In: International Workshop on Systems, Architectures, Modeling, and Simulation. (2003) 42–45
12. Stolberg, H.J., Berekovic, M., Pirsch, P.: A Platform-Independent Methodology for Performance Estimation of Streaming Media Applications. In: Proceedings 2002 IEEE International Conference on Multimedia and EXPO (ICME2002). (2002) CDROM.
13. Krashincky, R., Batten, C., Hampton, M., Gerding, S., Pharris, B., Casper, F., Asanovic, K.: The Vector-Thread Architecture. IEEE Micro **24** (2002) 84–90
14. Rixner, S., et al.: A Bandwidth-Efficient Architecture for Media Processing. In: Proceedings of 31st. Annual IEEE/ACM International Symposium on Microarchitectures MICRO-31. (1998) 3–13
15. Omondi, A.R.: Computer Arithmetic Systems: Algorithms, Architecture and Implementations. Prentice Hall International Series in Computer Science. (1994)
16. Lee, R.: Efficiency of MicroSIMD Architectures and Index-mapped Data for Media Processors. In: Proceedings of Media Processors. (1999) 34–46
17. Hennessy, J., Patterson, D.: Computer Architecture. A Quantitative Approach. 3rd edn. Morgan Kaufmann (2003)
18. Berekovic, M., Stolberg, H.J., Pirsch, P.: Multi-Core System-On-Chip Architecture for MPEG-4 Streaming Video. IEEE Transactions on Circuits and Systems for Video Technology (CSVT) **12** (2002) 688–699
19. Haque, F., Khan, K., Michelson, J.: The Art of Verification with VERA. Verification Central (2001)
20. Synopsys: Vera User Guide. (2003) version 6.0.
21. Gschwind, M., Salapura, V.: A VHDL Design Methodology for FPGAs. In: Field-Programmable Logic and Applications (FPL95). Volume 975 of Springer-Verlag Lecture Notes in Computer Science. (1995) 208–217
22. ProDesign Electronic GmbH: HDL Bridge and Signal Tracker. (2004)
23. Xilinx: Xilinx XAPP176 Configuration and Readback of the Spartan-II and Spartan-IIE Families Application Note. (2002) version 2.7.

Data-Driven Regular Reconfigurable Arrays: Design Space Exploration and Mapping

Ricardo Ferreira[1,⋆], João M.P. Cardoso[2,3], Andre Toledo[1], and Horácio C. Neto[3,4]

[1] Departamento de Informática,
Universidade Federal de Viçosa, Viçosa 36570 000, Brazil
cacau@dpi.ufv.br
[2] Universidade do Algarve, Campus de Gambelas, 8000-117, Faro, Portugal
jmpc@acm.org
[3] INESC-ID, 1000-029, Lisboa, Portugal
[4] Instituto Superior Técnico, Lisboa, Portugal
hcn@inesc.pt

Abstract. This work presents further enhancements to an environment for exploring coarse grained reconfigurable data-driven array architectures suitable to implement data-stream applications. The environment takes advantage of Java and XML technologies to enable architectural trade-off analysis. The flexibility of the approach to accommodate different topologies and interconnection patterns is shown by a first mapping scheme. Three benchmarks from the DSP scenario, mapped on hexagonal and grid architectures, are used to validate our approach and to establish comparison results.

1 Introduction

Recently, array processor architectures have been proposed as extensions of microprocessor based systems (see, e.g., [1], [2]). Their use to execute streaming applications leads to acceleration and/or energy consumption savings, both important for today and future embedded systems. Since many design decisions must be taken in order to implement an efficient architecture for a given set of applications, environments to efficiently experiment with different architectural features are fundamental. Array architectures may rely on different computational models. Architectures behaving in a static dataflow fashion [3][4] are of special interest, as they naturally process data streams, and therefore provide a very promising solution for stream-based computations, which are becoming predominant in many application areas [5]. In addition, the control flow can be distributed and can easily handle data-streams even in the presence of irregular latency times. In the data-driven model, synchronization can be achieved by ready-acknowledge protocols, the centralized control units are not needed, and the operations are dynamically scheduled by data flow. Furthermore, array architectures are scalable due to the regular design and symmetric structure connections. Moreover, high parallelism, energy

[⋆] Ricardo Ferreira acknowledges the financial support from Ct-Energia/CNPq, CAPES and FAPEMIG, Brazil.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 41–50, 2005.

consumption savings, circuit reliability and a short design cycle can be also reached by adopting reconfigurable, regular, data-driven array architectures [6]. However, many array architectures seem to be designed without strong evidences for the architectural decisions taken. Remarkably the work presented in [7][8] has been one of the few exceptions which addressed the exploration of architectural features (in this case, a number of KressArray [4] properties).

Our previous work presented a first step to build an environment to test and simulate data-driven array architectures [9]. To validate the concept we have presented results exploiting the size of input/output FIFOs for a simple example. As shown, the simulations are fast enough to allow the exploration of a significant number of design decisions. Our work aims to support a broad range of data-driven based arrays, a significant set of architecture parameters, and then evaluate its trade-offs using representative benchmarks. The environment will help the designer to systematically investigate different data-driven array architectures (topologies and connection patterns), as well as internal PE parameters (existence of FIFOs in PE input/outputs and their size, number of input/outputs of each PE, pipeline stages in each PE, etc.), and to conduct experiments to evaluate a number of characteristics (e.g., protocol overhead, node activity, etc.). An environment capable to exploit an important set of features is of great interest since it can provide an important aid on the design of new data-driven array architectures suitable to execute a set of kernels for specific application domains. The main contributions of this paper are:

- the integration in the environment of a first mapping scheme;
- the attainment of mapping results on grid and hexagonal arrays for three DSP benchmarks;

This paper is structured as follows. The following section briefly introduces the environment. Section 3 explains the mapping scheme. Section 4 shows examples and experimental results. Finally, section 5 concludes the paper and discusses ongoing and future work.

2 The Environment

A global view of our design exploration environment is shown in Fig. 1. The start point is the dataflow specification[1] which is written in XML. XML is also used to specify the coarse-grained, data-driven, array architecture, and the placement and routing. Each dataflow operator is directly implemented with a Functional Unit (FU). The environment uses Java to specify each FU behavior and to perform the dataflow and array modeling, simulation and mapping. For simulating either the array architecture or the specific design, we use the Hades simulation tool [10], which has been extended with a data-driven library. Note that we are interested on modeling and exploring data-driven array architectures in which, for a specific implementation of an algorithm, the PE operations

[1] A dataflow model can be automatically generated by a compiler from the input program in an imperative programming language [18][17].

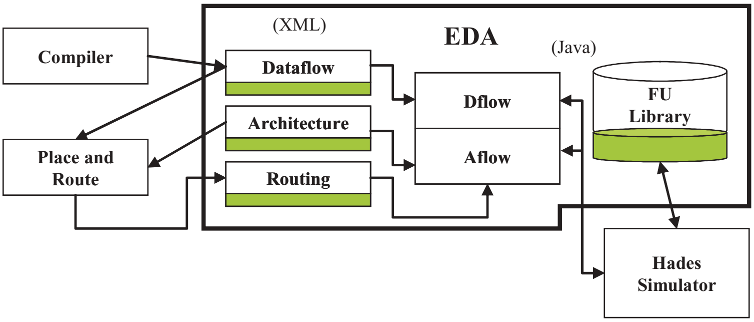


Fig. 1. Environment for Exploration of Data-Driven Array Architectures (EDA). Note that the Place and Route phase still needs further work and the front-end compiler is planned as future work

and interconnections between them are statically defined[2]. Our environment supports two simulation flows (Dflow and Aflow in Fig. 1):

- In Dflow, the dataflow representation is translated to a Hades Design and simulated. Dflow provides an estimation of the optimal performance (e.g., achievable when implementing an ASIC based architecture) provided that full balancing is used (i.e., FIFOs of enough size are used). It permits a useful comparison with implementations in a reconfigurable data-driven array, since it represents the optimal achievable performance using a specific set of FUs (akin to the FUs existent in each PE of the array).
- In Aflow, the dataflow representation is mapped to a data-driven array architecture, specified by a template, and is simulated with Hades. For design analysis, an user may specify, in the dataflow and array architecture descriptions, which parameters should be reported by the simulator engine. Those parameters can be the interconnect delay, the handshake protocol overhead, the operator activity, etc. As some experimental results show, the simulation and the mapping is fast enough to conduct a significant number of experiments.

A typical FU integrates an ALU, a multiplier or divider, input/output FIFOs, and the control unit to implement the handshake mechanism (see Fig. 2a). The FU is the main component of each PE in the array architecture. A PE consists on an FU embedded on an array cell, which has a regular neighbor pattern implemented by local interconnections (see Fig. 2b and Fig. 2c). A ready/acknowledge based protocol controls the data transfer between FUs or PEs. An FU computes a new value when all required inputs are available and previous results have been consumed. When an FU finishes computation, an acknowledge signal is sent back to all inputs and the next data tokens can be received. Each FU, besides traditional data-driven operators [3], may also implement the SE-PAR and PAR-SE operators introduced in [12] [13]. These operators resemble

[2] TRIPS [11] is an example of an architecture with interconnections dynamically defined.

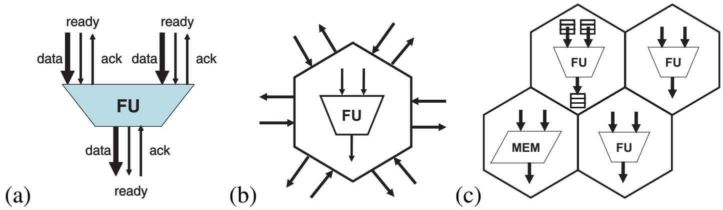


Fig. 2. (a) Data-driven functional unit (FU) with two inputs and one output; (b) Hexagonal cell with the FU; (c) Hexagonal array (FUs may have FIFOs in their input/output ports)

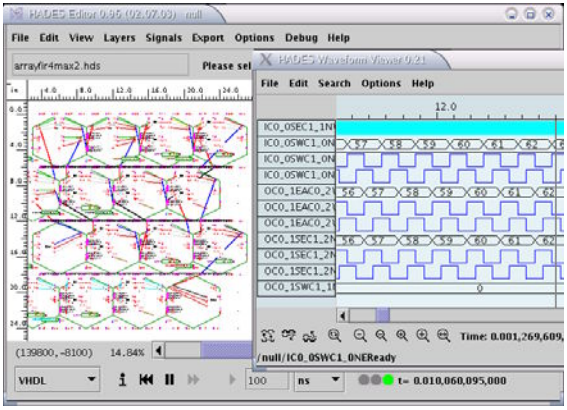


Fig. 3. A hexagonal array interactive simulation using Hades

mux and demux operators without external control. The canonical form of PAR-SE has two inputs (A and B) and one output (X). It repeatedly outputs to X the input in either A and B, in an alternating fashion. The canonical form of SE-PAR has one input (A) and two outputs (X and Y). The operator repeatedly alternates data on the input to either X or Y. Note, however, that PAR-SE and SE-PAR can have more than two inputs and two outputs, respectively. They can be used to reduce array resources and to fully decentralize the control structure needed. These operations provide an efficient way of sharing resources whenever needed (e.g., interface to an input/output port, interface to memory ports, etc.). SE-PAR and PAR-SE operations with more than two outputs and two inputs, respectively, can be implemented by a tree of basic SE-PAR and PAR-SE.

Each FU may have input/output FIFOs, which can be efficient structures to handle directly unbalanced paths. Parameters such as protocol delay, FIFO size, FU granularity are global in the context of an array but can be local when a specific dataflow implementation is the goal. At this level, an FU behavior and the communication protocol are completely independent of the array architecture. They are specified as Java classes, which provide an easy way to write an incremental FU library and then to model and to simulate a pure dataflow, as well as a novel architecture.

The properties of the target architecture such as the array topology, the interconnection network, and the PE's placement, are specified using XML-based languages, which provide an efficient way to explore different array architectures.

We can use the graphical user interface of Hades to perform interactive simulation at Dflow or Aflow. Fig. 3 shows a screenshot with a hexagonal array simulation and the respective waveforms.

3 A Flexible Mapping Approach

An array processor can be defined as a set of PEs connected with different topologies. Our goal is to allow the exploration of data-driven architectures, which can have a mesh, a hexagonal or any other interconnection network. Previous works [14][15] have addressed the mapping of data-driven algorithms on regular array architectures. A mapping algorithm for a regular hexagonal array architecture has been proposed in [14], with a fixed interconnection degree. On the other hand, most array architectures are based on grid topology [15]. Our approach differs from the previous ones in two significant ways: (a) it is currently able to compare hexagonal and grid topologies; (b) it presents an object-oriented mapping scheme to model different patterns; (c) it is flexible enough to accommodate other mapping algorithms.

Our object-oriented mapping scheme also takes advantages of Java and XML technologies to enable a portable and flexible implementation. The scheme provides an easy way of modeling grid, hexagonal, octal, as well as others topologies. The implementation is based on three main classes: the *Array*, the *PE*, and the *Edge*. The *Array* class implements the place and routing algorithm. The array and neighbor parameters (e.g., number and their positions) are specified using *PE* classes. Finally, the *Border* class models the number and type of connections between neighbors. Examples of *PE* and *Edge* classes are represented in Fig. 4. Each *PE* defines the number of borders with the neighbors, with each border having the input/output connections defined by the *Edge* class. At the moment the scheme does not accept different *Edges*.

A PE can be connected to N-hop neighbors (see Fig. 5a) and can have in, out and/or in-out connections (see Fig. 5b). In a 0-hop pattern, each PE is connected to the immediate neighbors. In a 1-hop pattern, each PE is connected to the immediate neighbors and 1-hop, i.e., the PEs that can be reached by traversing through one neighbor PE. For instance, the nodes 1 and 3 are 1-hop neighbors in Fig. 5a.

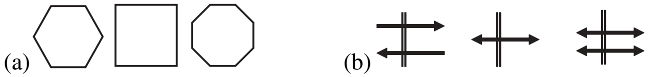


Fig. 4. Main classes of the mapping scheme: (a) PE classes, each one with different edges; (b) Edge classes (three parameters are used: number of input, output, and input/output connections

Two versions of a first mapping algorithm have been developed (versions PR1 and PR2). They are based on the greedy algorithm presented in [14]. Albeit simple, they enabled us to explore different connection patterns and different topologies. The mapping

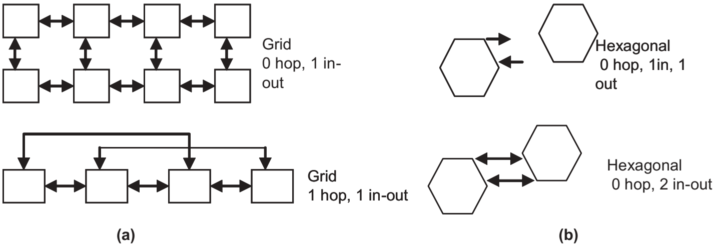


Fig. 5. Different topologies supported by the mapping phase: (a) 0-hop and 1-hop Grid Topology (b) Uni-directional and Bi-directional Neighbor Connection

algorithm is divided in place and route steps. The algorithm starts by placing the nodes of the dataflow graph (i.e., assign a PE for each node of the graph) based on layers and then optimizes the placement based on the center mass forces. The PR1 version adds NOP nodes in the input dataflow graph before starting the optimization phase. After placement, the route step tries to connect the nodes using incremental routing (e.g., each path of the original DFG is constructed by routing from one PE to one of its neighbors).

The infrastructure has been developed to easily integrate different mapping algorithms. Future work will address a more advanced mapping algorithm. We plan to add critical path sensibility and to include path balancing, which can be very important in array architectures without or with small size FIFOs. A scheme to deal with heterogeneous array elements (e.g., only some PEs with support for multiplication) should also be researched. Notice also that the arrays currently being explored do not permit the use of a PE to implement more than one operation of the DFG. Arrays including this feature require node compression schemes as has been used in [14] for hexagonal arrays.

4 Experimental Results

We have used the current prototype environment to perform some experiments. In the examples presented, we have used 32-bit width FUs and a 4-phase asynchronous handshake mechanism. All executions and simulations have been done in a Pentium 4 (at 1.8 GHz, 1 GB of RAM, with Linux).

4.1 Examples

As benchmarks we use three DSP algorithms: FIR, CPLX, and FDCT. FIR is a finite-impulse response filter. CPLX is a FIR filter using complex arithmetic. FDCT is a fast discrete cosine transform implementation. The last two benchmarks are based on the C code available in [16]. For the experiments, we manually translated the input algorithms to a dataflow representation. The translation has been done bearing in mind optimization techniques that can be included in a compiler from a software programming language (e.g., C) to data-driven representations (see, e.g., [17][18] for details about compilation issues).

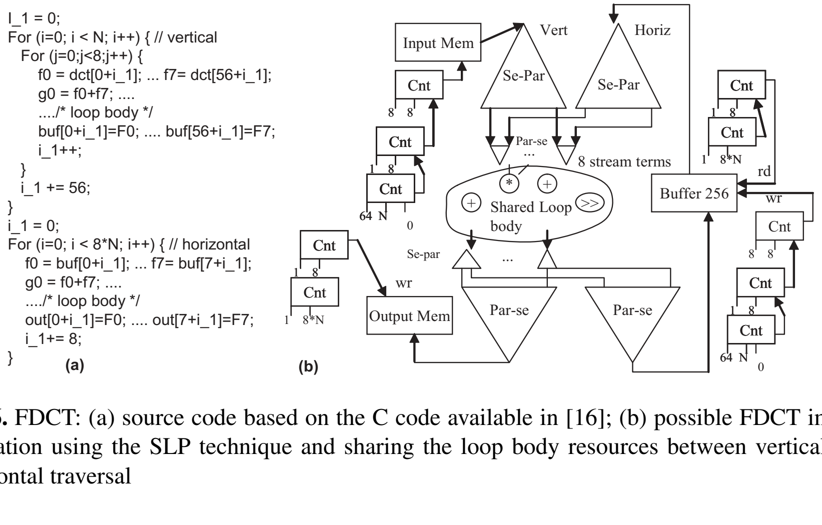
Table 1. Properties related to the implementations of three DSP benchmarks

Ex	#FU	#copy	#ALU	#MULT	#SE-PAR	#PAR-SE	#I/O	Average activity (ALU+MULT)	Average activity (all)	Max (ALU+MULT)	Max ILP (all)
FIR-2	7	1	2	2	0	0	2	1.00	1.00	4	7
FIR-4	13	3	4	4	0	0	2	1.00	1.00	8	13
FIR-8	25	7	8	8	0	0	2	1.00	1.00	16	25
FIR-16	49	15	16	16	0	0	2	1.00	1.00	32	49
CPLX4	22	5	8	2	4	1	2	0.70	0.86	8	18
CPLX8	46	13	18	4	8	1	2	0.68	0.82	16	38
FDCTa	92	26	36	14	7	7	2	0.12	0.18	10	23
FDCTb	102	26	46	14	7	7	2	0.12	0.16	12	25
FDCT	136	26	52	14	21	21	2	0.20	0.26	22	49

For the data-driven implementation of the FDCT example (see part of source code in Fig. 6a) we used the *SLP* (self loop pipelining) technique with SE-PAR and PAR-SE operators [13][12]. See the block diagram in Fig. 6b. An SE-PAR tree splits the matrix input stream in 8 parallel elements. Then, the inner loop operations are performed concurrently, and finally, a PAR-SE tree merges the results into a unique matrix output stream. Notice that the SE-PAR and PAR-SE operators are also used here to share the computational structures of the two loops of the FDCT.

4.2 Results

Table 1 shows the number of resources needed (number of FUs) and the results obtained by simulating implementations of the FIR and CPLX filters, with different number of taps, and two parts of the FDCT (FDCTa is related to the vertical traversal and FDCTb is related to the horizontal traversal) and the complete FDCT. In these experiments FIFOs (size between 1 and 3) in the inputs and outputs of each FU are used to achieve the maximum throughput. The CPU time to perform each simulation has been between 1 to 4 seconds for 1,024 input data items.



```
I_1 = 0;
For (i=0; i < N; i++) { // vertical
  For (j=0;j<8;j++) {
     f0 = dct[0+i_1]; ... f7= dct[56+i_1];
     g0 = f0+f7; ....
     ..../* loop body */
     buf[0+i_1]=F0; .... buf[56+i_1]=F7;
     i_1++;
  }
  i_1 += 56;
}
i_1 = 0;
For (i=0; i < 8*N; i++) { // horizontal
   f0 = buf[0+i_1]; ... f7= buf[7+i_1];
   g0 = f0+f7; ....
   ..../* loop body */
   out[0+i_1]=F0; .... out[7+i_1]=F7;
   i_1+= 8;
}
```

Fig. 6. FDCT: (a) source code based on the C code available in [16]; (b) possible FDCT implementation using the SLP technique and sharing the loop body resources between vertical and horizontal traversal

The average activity columns show the percentage of time in which an FU performs an operation. The maximum activity (i.e., 1.00) is reached when the FU activity is equal to the input stream rate. We present average activities taking into account only ALU+MULT operations and all the operations. The maximum ILP (instruction level parallelism) shows the maximum number of FUs executing at a certain time step. Once again, we present ILP results for ALU+MULT and for all operations. As we can see for FIR and CPLX the maximum ILP is approximately equal to the number of FUs, which depicts that all the FUs are doing useful work almost all the time. With FDCT, the maximum ILP is high (22 for ALU+MULT operations and 49 for all operations, considering the complete example) but many FUs are used only small fractions of the total execution time. We believe this can be improved by using SE-PAR and PAR-SE operators with more outputs and inputs, respectively. For instance, in the hexagonal array we may have implementations of these operators with 6 inputs or 6 outputs, which significantly reduce the SE-PAR and PAR-SE trees. Fig. 7 shows a snapshot of the execution of FDCT showing the operations activity.

The mapping algorithms presented in Section 3 have been implemented in Java. Table 2 shows the mapping results for the three benchmarks on three different array topologies (Grid, Grid 1-hop, and Hexagonal), each one with two different connection structures (0,0,2 and 2,2,0 indicate 2 bidirectional connections, and 2 input and 2 output connections in each *Edge* of a PE, respectively). Each example has been mapped in 200 to 400 ms of CPU time.

Column "N/E" specifies the number of nodes and edges for each dataflow graph. Average path lengths (measured as the number of PEs that a connection needs to traverse from the source to the sink PE) after mapping the examples onto three topologies are shown in columns "P". Columns "M" represent the maximum path length for each example when mapped in the correspondent array. Cells in Table 2 identified by "-" represent cases unable to be placed and routed by the current version of our mapping algorithm. Those cases happened with the FDCT examples on the Grid topology.

The results obtained, using the implemented mapping scheme, show that the simpler Grid topology is the worst in terms of maximum and average path lengths. Hexagonal and the Grid 1-hop perform distinctly according to the benchmark. The hexagonal seems

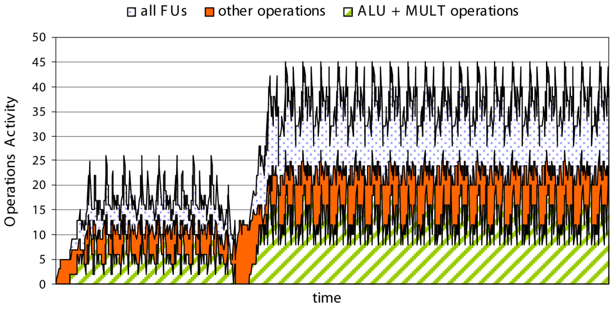


Fig. 7. Activity of the FUs during initial execution of the FDCT implementation shown in Fig. 6b. After 102 input samples the implementation starts outputting each result with maximum throughput

Table 2. Results of mapping the benchmarks on different array topologies and interconnection patterns between PEs

EX	N/E	P&R	Grid				Grid 1-hop				Hexagonal			
			0,0,2		2,2,0		2,2,0		0,0,2		2,2,0		0,0,2	
			P	M	P	M	P	M	P	M	P	M	P	M
FIR2	7/7	PR1	**1.28**	**2**	**1.28**	**2**	**1.28**	**2**	**1.28**	2	**1.28**	**2**	**1.28**	**2**
		PR2	1.28	**2**	1.28	**2**	1.28	**2**	1.28	**2**	**1.14**	**2**	**1.14**	**2**
FIR4	13/15	PR1	1.86	4	1.60	3	1.40	2	1.40	**2**	**1.33**	**2**	**1.33**	**2**
		PR2	1.60	3	1.60	3	1.46	**2**	1.46	**2**	**1.26**	**2**	**1.26**	**2**
FIR8	25/31	PR1	1.96	6	2.03	5	1.58	3	1.58	**3**	**1.54**	4	**1.54**	4
		PR2	1.83	5	1.83	5	1.54	3	1.54	**3**	**1.51**	4	**1.51**	4
FIR16	49/63	PR1	2.25	9	2.26	11	1.71	**5**	1.69	**5**	**1.55**	7	**1.55**	7
		PR2	2.19	9	2.15	11	**1.71**	**5**	1.73	**5**	**1.71**	8	**1.71**	8
CPLX4	22/28	PR1	1.71	6	1.75	6	1.46	**3**	1.46	**3**	**1.39**	5	**1.39**	5
		PR2	1.71	6	1.71	6	**1.46**	**3**	**1.46**	**3**	1.50	4	1.50	4
CPLX8	46/60	PR1	2.46	10	2.31	11	**1.73**	**5**	**1.73**	**5**	1.75	7	1.76	7
		PR2	2.13	10	2.21	11	**1.61**	**6**	**1.61**	**6**	1.80	8	1.80	8
FDCTa	92/124	PR1	-	-	2.49	14	**1.83**	**6**	2.09	**7**	2.08	8	2.32	9
		PR2	-	-	2.41	10	1.83	5	**1.96**	10	2.07	10	2.10	9
FDCTb	102/134	PR1	-	-	2.32	14	**1.75**	6	1.94	**7**	2.01	10	2.24	9
		PR2	-	-	2.42	12	1.76	**5**	**1.85**	8	1.97	10	2.02	10
FDCT	136/186	PR1	-	-	-	-	3.19	15	3.31	21	4.61	22	4.31	28
		PR2	-	-	-	-	**2.91**	**13**	**3.01**	**16**	3.71	20	4.04	21

to perform better for the FIR filters and is outperformed by the Grid 1-hop for the other benchmarks (CPLX and FDCT). Values in bold in Table 2 highlight the best results. The results confirm that the Grid 1-hop outperforms the Grid topology as been already shown in [15]. Note however that the hexagonal topology was not evaluated in [15].

5 Conclusions

This paper presents further enhancements in an environment to simulate and explore data-driven array architectures. Although in those architectures many features are worth to be explored, developing an environment capable to exploit all the important features is a tremendous task. In our case we have firstly selected a subset of properties to be modeled: FIFO sizes, grid or hexagonal topologies, etc. Notice, however, that the environment has been developed bearing in mind incremental enhancements, each one contributing to a more powerful exploration.

A first version of a mapping approach developed to easily explore different array configurations is presented and results achieved for hexagonal and grid topologies are shown. This first version truly proves the flexibility of the scheme.

Ongoing work intends to add more advanced mapping schemes to enable a comparison between different array topologies independent from the mapping algorithm used to conduct the experiments. Forms to deal with heterogeneous array elements distributed through an array are also under focus.

Further work is also needed to allow the definition of the configuration format for each PE of the architecture being evaluated, as well as, automatic VHDL generation to prototype a certain array or data-driven solution in an FPGA. We have also long-term plans to include a front-end compiler to continue studies of some data-driven array features with complex benchmarks.

We really hope that further developments will contribute to an environment able to evaluate new data-driven array architectures prior to fabrication.

References

1. Hartenstein, R.: A Decade of Reconfigurable Computing: a Visionary Retrospective. In: Int'l Conf. on Design, Automation and Test in Europe (DATE'01), Munich, Germany (2001) 642–649.
2. Bossuet, L., Gogniat, G., Philippe, J.L.: Fast design space exploration method for reconfigurable architectures. In: Int'l Conference on Engineering of Reconfigurable Systems and Algorithm (ERSA'03), Las Vegas, Nevada (2003)
3. Veen, A.H.: Dataflow machine architecture. ACM Computing Surveys **18** (1986) 365–396
4. Hartenstein, R., Kress, R., Reinig, H.: A Dynamically Reconfigurable Wavefront Array Architecture. In Proc. Int'l Conference on Application Specific Array Processors (ASAP'94) (1994) 404–414
5. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A Language for Streaming Applications. In: Proc. of the Int'l Conf. on Compiler Construction (CC'02) (2002)
6. Imlig, N., et al.: Programmable Dataflow Computing on PCA. IEICE Trans. Fundamentals **E83-A** (2000) 2409–2416
7. Hartenstein, R., Herz, M., T. Hoffmann, T., Nageldinger, U.: Generation of Design Suggestions for Coarse-Grain Reconfigurable Architectures. In: 10th Int'l Workshop on Field Programmable Logic and Applications (FPL'00), Villach, Austria (2000)
8. R. Hartenstein, R., Herz, M., Hoffmann, T. Nageldinger, U.: KressArray Xplorer: A New CAD Environment to Optimize Reconfigurable Datapath Array Architectures. In: 5th Asia and South Pacific Design Automation Conference (ASP-DAC'00), Yokohama, Japan (2000) 163–168
9. Ferreira, R., Cardoso, J.M.P., Neto, H.C.: An Environment for Exploring Data-Driven Architectures. In: 14th Int'l Conference on Field Programmable Logic and Applications (FPL'04), LNCS 3203, Springer-Verlag (2004) 1022-1026
10. Hendrich, N.: A Java-based Framework for Simulation and Teaching. In: 3rd European Workshop on Microelectronics Education (EWME'00), Aix en Provence, France (2000) 285–288
11. Burger, D., et al.:Scaling to the End of Silicon with EDGE architectures. IEEE Computer (2004) 44–55
12. Cardoso, J.M.P: Self Loop Pipelining and Reconfigurable Dataflow Arrays. In: Int'l Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS IV), Samos, Greece, LNCS 3133, Springer Verlag (2004) 234–243
13. Cardoso, J.M.P.: Dynamic Loop Pipelining in Data-Driven Architectures. In: ACM Int'l Conference on Computing Frontiers (CF'05), Ischia, Italy (2005)
14. Koren, I., et al.: A Data-Driven VLSI Array for Arbitrary Algorithms. IEEE Computer **21** (1989) 30–43
15. Bansal, N., et al.: Network Topology Exploration of Mesh-Based Coarse-Grain Reconfigurable Architectures. In: Design, Automation and Test in Europe Conference (DATE '04), Paris, France (2004) 474–479
16. Texas Instruments, Inc. TMS320C6000 Highest Performance DSP Platform. 1995-2003, http://www.ti.com/sc/docs/products/dsp/c6000/benchmarks/62x.htm#search
17. Budiu, M., Goldstein, S.C.: Compiling application-specific hardware. In: Proceedings 12th Int'l Conference on Field Programmable Logic and Applications (FPL'02), LNCS 2438, Springer-Verlag (2002) 853–863
18. Cardoso, J.M.P, Weinhardt, M.: XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture. In: 12th Int'l Conference on Field Programmable Logic and Applications (FPL'02), LNCS 2438, Springer Verlag (2002) 864–874

Automatic FIR Filter Generation for FPGAs

Holger Ruckdeschel, Hritam Dutta, Frank Hannig, and Jürgen Teich

Department of Computer Science 12, Hardware-Software-Co-Design,
University of Erlangen-Nuremberg, Germany
{dutta, hannig, teich}@cs.fau.de

Abstract. This paper presents a new tool for the automatic generation of highly parallelized Finite Impulse Response (FIR) filters. In this approach we follow our PARO design methodology. PARO is a design system project for modeling, transformation, optimization, and synthesis of massively parallel VLSI architectures. The FIR filter generator employs during the design flow the following advanced transformations, (a) *hierarchical partitioning* in order to balance the amount of local memory with external communication, and (b), *partial localization* to achieve higher throughput and smaller latencies. Furthermore, our filter generator allows for design space exploration to tackle trade-offs in cost and speed. Finally, synthesizable VHDL code is generated and mapped to an FPGA, the results are compared with a commercial filter generator.

1 Introduction and Related Work

At all times, there was a need for high performance, low size, low cost, low power, and other criteria, and therefore the ambition to develop dedicated massively parallel hardware to achieve these goals. We can not imagine life today without audio, video, and speech communication which are all based on digital signal processing. The applications considered in this variety of areas are well qualified for hardware implementation because of their inherent data parallelism and the common computational units found in most signal processing algorithms. A lot of these applications can be described by algorithm classes based on sets of recurrence equations which are closely related to *single assignment code* (SAC), where the whole parallelism is explicitly given. However only few synthesis tools for the design of application specific circuits exists when starting from a given nested loop program. For instance, Compaan [1] which deals with process networks or PICO-N initially developed by the Hewlett-Packard Laboratories [2,3] and recently commercialized as PICO Express by Synfora [4]. PARO [5, 6], is a design system project for modeling, transformation, optimization, and processor synthesis for the class of *Piecewise Linear Algorithms* (PLA). PARO can be used for the process of automated synthesis of regular circuits and will be later described in this paper. Certainly, there exists a number of fully developed hardware design languages and tools like Handel-C [7] but they use imperative forms as input code or they do not allow high-level program transformations.

In this paper we present a flexible design tool for the automatic synthesis of FIR filters. In general, digital filters are used for two purposes, (1) for the separation of signals that have been combined, and (2), the restoration of signals that have been distorted in some way, i.e., filters modify or remove unwanted frequencies from an input signal. Since FIR filters have a wide field of application there exist quite a large number

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 51–61, 2005.

of tools for designing and generating filters. For example, in [8] the authors describe an approach for the automatic VHDL generation of FIR filters by the use of truncated multipliers in order to decrease the design complexity. Other approaches to reduce the filter size are distributed arithmetic as used in Xilinx CORE Generator [9] or in Altera's FIR Compiler [10]. Closely related to the basic concepts presented here is the approach in [11] where the authors use MMAlpha for the automatic generation of pipelined LMS adaptive filters.

Beside the possibilities to parameterize the bitwidths of input and output signals, and the number of taps, the following aspects are the novel contributions of this paper:

– Highly parallel implementation as a *two-dimensional pipelined processor array*.
– Application of *hierarchical partitioning techniques* in order to balance the amount of local memory with external communication [12].
– Usage of *partial localization* [13] to achieve throughput higher than 100% (real parallel computation) and smaller latencies.

The rest of the paper is structured as follows. In Section 2, we introduce briefly some basic definitions and transformations for the automatic parallelization in the polytope model. In Section 3 and 4, the architecture design and features of our developed *Firgen* tool are described. In Section 5, results and comparison of the generated FIR filters to state of the art tools are presented. In Section 6, we conclude our work.

2 Definitions, Notations, and Some Concepts

The class of algorithms dealt with in our methodology is a class of recurrence equations known as Piecewise Linear Algorithms (PLA) [13].

Example 1. The FIR (Finite Impulse Response) filter is described by the simple difference equation $y(i) = \sum_{j=0}^{N-1} a(j) \cdot u(i-j)$ with $0 \leq i < T$, N denoting the number of filter taps, $a(j)$ the filter coefficients, $u(i)$ the filter input, and $y(i)$ the filter result. The difference equation on *parallelization* and *embedding* in a common index space can be written as the following PLA

$$\begin{aligned} a[i,j] &= a[0,j]; \quad u[i,j] = u[i-j,0]; \quad x[i,j] = a[i,j] \cdot u[i,j]; \\ y[i,j] &= y[i,j-1] + x[i,j]; \end{aligned} \tag{1}$$

with the iteration domain $I = \{(i,j) \mid 0 \leq i \leq T-1 \wedge 0 \leq j \leq N-1\}$.

The process of obtaining a hardware description in VHDL from PLA definitions involves *scheduling* and *allocation* transformations (see Section 2.2) to obtain a full-size processor array (PA) implementation. However, full size implementations are dependent on problem parameters. Therefore another well known transformation *partitioning* is used to obtain reduced size PAs which meet the architecture constraints. *Localization* is a transformation to convert broadcast signals into short propagation links. The localization of all input and output variables gives the following set of recurrence equations for the FIR filter.

$$\begin{aligned} a[i,j] &= a[i-1,j]; \quad u[i,j] = u[i-1,j-1]; \quad x[i,j] = a[i,j] \cdot u[i,j]; \\ y[i,j] &= y[i,j-1] + x[i,j]; \end{aligned} \tag{2}$$

In the next subsections, we describe the mentioned transformations in detail.

2.1 Partitioning

Partitioning is a transformation which covers the index space of computation using congruent hyperplanes, hyperquaders, or parallelepipeds called *tiles*. The transformation has been studied in detail for compilers and its use has led to program acceleration through better cache reuse on sequential processors (i.e., *loop tiling*) [14], implementation of algorithms on given parallel architectures from supercomputers to multi-DSPs and FPGAs. For PAs, it is carried out in order to match a loop nest implementation to resource constraints. Well known partitioning techniques are multiprojection, LSGP (local sequential global parallel, often also referred as clustering or blocking) and LPGS (local parallel global sequential, also referred as tiling). Formally, partitioning divides the index space I using congruent tiles such that it is decomposed into spaces $\mathcal{J}$ and $\mathcal{K}$, i.e., $I \mapsto \mathcal{J} \oplus \mathcal{K}$. $\mathcal{J} \in \mathbb{Z}^n$ represents the points within the tile and $\mathcal{K} \in \mathbb{Z}^n$ accounts for regular repetition of the tiles, i.e., the origin of each tile. Hierarchical partitioning methods use different hierarchies of tiling matrices to divide the index space. Co-partitioning is such an example of a 2-level hierarchical partitioning [12], where the index space is first partitioned into LS (local sequential) tiles, this tiled index space is tiled once more using GS (global sequential) tiles as shown in Fig. 1. Co-partitioning uses both LSGP and LPGS methods in order to balance local memory requirements with I/O bandwidth with the advantage of problem size independence. Formally, it is defined as splitting of

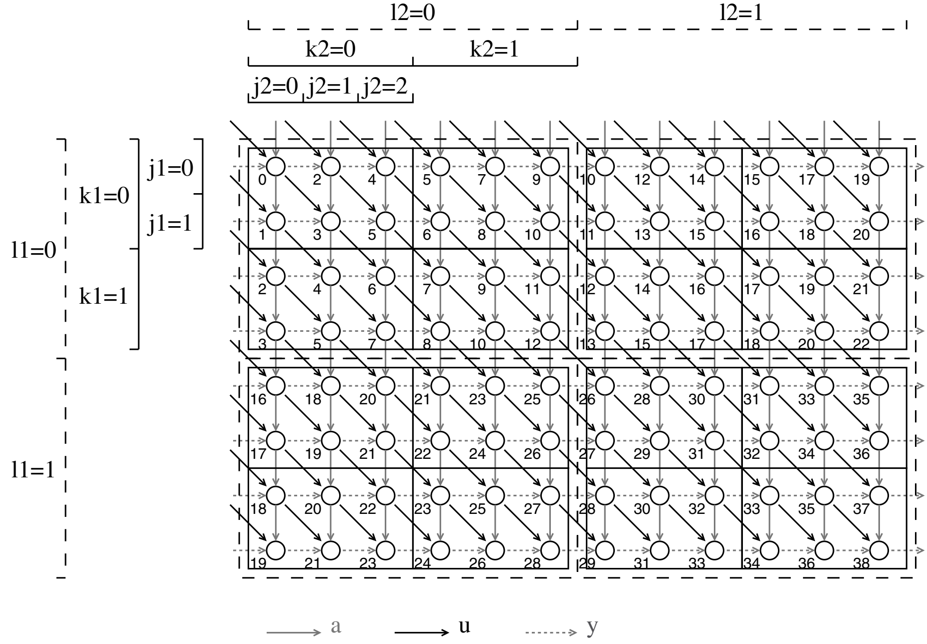


Fig. 1. The iteration space of an example localized co-partitioned FIR filter multiplication with 12 taps. Each arc denotes a data dependency

an index space into spaces $\mathcal{J}$, $\mathcal{K}$ and $\mathcal{L}$, i.e., $I \mapsto \mathcal{J} \oplus \mathcal{K} \oplus \mathcal{L}^1$ using two congruent tile types defined by tiling matrices, P_{LS} and P_{GS}. $\mathcal{J} \in \mathbb{Z}^n$ represents the points within the LS tiles and $\mathcal{K} \in \mathbb{Z}^n$ accounts for the regular repetition of the origin of LS tiles (i.e., tiles marked with solid line in Fig. 1). $\mathcal{L} \in \mathbb{Z}^n$ accounts for the regular repetition of the GS tiles (i.e., bigger tiles marked with dotted line in Fig. 1). Different partitioning schemes such as LSGP, LPGS, and co-partitioning are defined by specific scheduling functions which are typically realized through appropriate affine transformations defining the allocation and scheduling (see Section 2.2).

Example 2. The dataflow graph of the localized co-partitioned FIR filter with tiling matrices $P_{LS} = \begin{pmatrix} J_1 & 0 \\ 0 & J_2 \end{pmatrix}$, $P_{GS} = \begin{pmatrix} K_1 & 0 \\ 0 & K_2 \end{pmatrix}$ with $J_1 = 2, J_2 = 3, K_1 = 2, K_1 = 2$ is shown in Fig. 1. The recurrence equation for only the weights, i.e., a on co-partitioning is as follows. For the sake of brevity the description of other variables (i.e. u, y) has been omitted.

$$a[j_1, j_2, k_1, k_2, l_1, l_2] = \begin{cases} a(j_2, k_2, l_2) & \text{if } j_1 = 0 \wedge k_1 = 0 \wedge l_1 = 0 \\ a[j_1 - 1, j_2, k_1, k_2, l_1, l_2] & \text{if } j_1 > 0 \\ a[j_1 + J_1 - 1, j_2, k_1 - 1, k_2, l_1, l_2] & \text{if } j_1 = 0 \wedge k_1 > 0 \\ a[j_1 + J_1 - 1, j_2, & \\ \quad k_1 + K_1 - 1, k_2, l_1 - 1, l_2] & \text{if } j_1 = 0 \wedge k_1 = 0 \wedge l_2 > 0 \end{cases}$$

The spaces after co-partitioning are $\mathcal{J} = \{J = (j_1, j_2) \mid 0 \leq j_1 < J_1 \ \wedge \ 0 \leq j_2 < J_2\}$, $\mathcal{K} = \{k_1, k_2 \mid 0 \leq k_1 < K_1 \ \wedge \ 0 \leq k_2 < K_2\}$, $\mathcal{L} = \{l_1, l_2 \mid 0 \leq l_1 < L_1 \ \wedge \ 0 \leq l_2 < L_2\}$

2.2 Space-Time Mapping

Linear transformations are used as *space-time mappings* in order to assign a processor p (space) and a sequencing index t (time) to index vectors [15]. In co-partitioning, the index points within the LS tiles are executed sequentially. All the LS tiles within a GS tile are executed in parallel by the PA. The GS tiles are executed sequentially.

Definition 1. *(Space-time mapping for co-partitioning). A space-time mapping in case of co-partitioning is an affine transformation of the form*

$$\begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} 0 & E & 0 \\ \lambda_J & \lambda_K & \lambda_L \end{pmatrix} \begin{pmatrix} J \\ K \\ L \end{pmatrix} \tag{3}$$

where $E \in \mathbb{Z}^{n_K \times n_K}$ is the identity matrix, $\lambda_J \in \mathbb{Z}^{1 \times n_J}$, $\lambda_K \in \mathbb{Z}^{1 \times n_K}$, $\lambda_L \in \mathbb{Z}^{1 \times n_L}$.

Similarly, other partitioning schemes can be realized using an appropriate selection of affine transformations characterizing the scheduling and the allocation of the index points. The problem of determining an optimal sequencing index (i.e., $\lambda_J, \lambda_K, \ldots$) taking into account constraints on timing of PAs and availability of resources might be solved by a Mixed Integer Linear Programming (MILP) formulation of the problem similar as in [16].

[1] $\mathcal{J} \oplus \mathcal{K} \oplus \mathcal{L} = \{i = j + P_{LS} \cdot k + P_{GS} \cdot l \mid j \in \mathcal{J} \ \wedge \ k \in \mathcal{K} \ \wedge \ l \in \mathcal{L} \ \wedge \ P_{LS}, P_{GS} \in \mathbb{Z}^{n \times n}\}$.

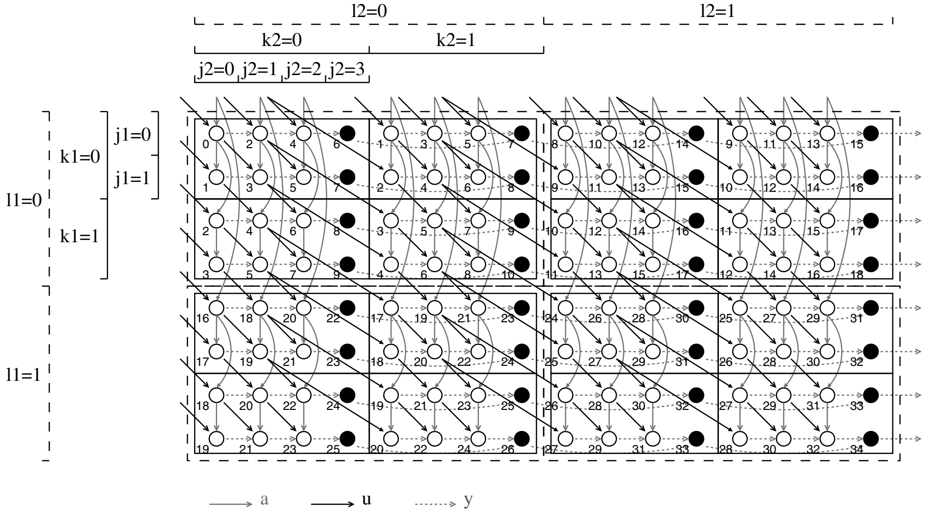


Fig. 2. Iteration space of the partially localized co-partitioned FIR filter with 12 taps. Each arc denotes a data dependency. The black points denote the partial sums to be added, introduced due to partial localization

Example 3. One can verify that scheduling and allocation for the dataflow graphs in Fig. 1 and Fig. 2 are given by the following space-time mappings.

$$\begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} 0\,0\,1\,0\,0 & 0 \\ 0\,0\,0\,1\,0 & 0 \\ 1\,2\,2\,5\,16 & 10 \end{pmatrix} \begin{pmatrix} J \\ K \\ L \end{pmatrix} \qquad \begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} 0\,0\,1\,0\,0 & 0 \\ 0\,0\,0\,1\,0 & 0 \\ 1\,2\,2\,1\,16 & 8 \end{pmatrix} \begin{pmatrix} J \\ K \\ L \end{pmatrix}$$

2.3 Partial Localization

Localization prior to partitioning introduces unnecessary copy operations and additionally restricts optimal schedules available after partitioning [13]. A new design flow implements partitioning before localization and therefore eliminates above mentioned disadvantages. The concept of partial localization entails localization of data dependencies for intra-tile dependencies in case of LPGS partitioning and inter-tile dependencies for LSGP partitioning [13], and partial localization for intermediate schemes. Fig. 2 shows an example of the partially localized dataflow graph of the FIR filter. The latency for the partially localized example is 15 cycles as compared to 19 cycles for the fully localized example.

3 Architecture Design

This section describes the architecture design of the Firgen implementation. First, an overview about the architecture is given and afterwards the major components of the design are described in detail.

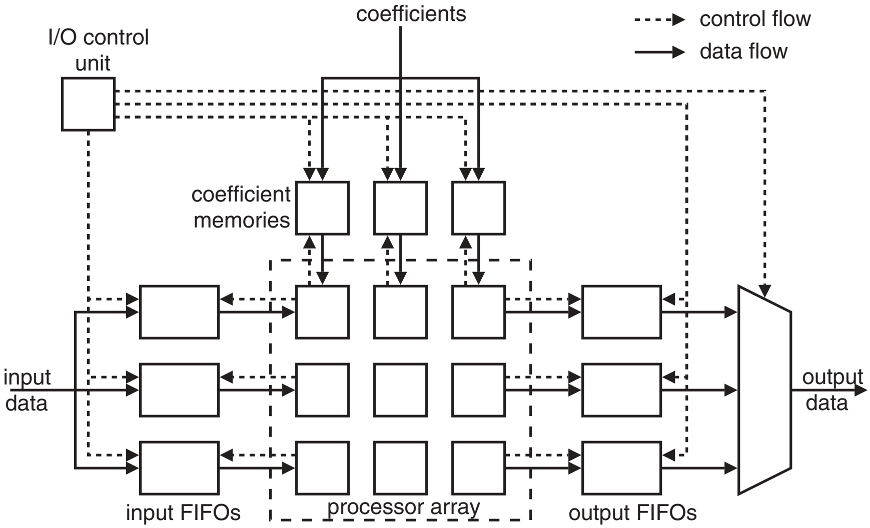


Fig. 3. Overview of the Firgen FIR filter architecture for a processor array with 3 × 3 processors

Architecture Overview. The core of the design is a 2-d array of $K_1 \times K_2$ processor elements (PE). Where K_1, K_2 can be selected according to FPGA architecture constraints. Each PE contains one fixed-point, full precision multiply-and-accumulate unit (MAC), which is used to compute the basic MAC operation in the FIR filter. This PA implements the co-partitioned FIR filter as described in Section 2, with either fully or partially localized data dependencies. An overview of the architecture of Firgen is shown in Fig. 3. The PA gets its input samples from a number of input FIFOs and stores the results in the output FIFOs, where each row of the PA has its own pair of I/O-FIFOs. The streaming input is distributed to corresponding input FIFOs as determined by an I/O control unit. Similarly, each column of the array reads its coefficients from a separate coefficient memory. In contrast to many other FIR filter implementations, like *Xilinx Coregen*, the coefficients need not to be specified at synthesis time, but are stored in a RAM, allowing the user to change the coefficients.

Processor Array. The structure of the processor array is depicted in Fig. 4. For each data dependency crossing LS tiles, there is a corresponding interconnection between the respective processors, with the appropriate number of delay registers between them. Each processor needs the counter signals j_1, j_2, l_1, and l_2 to generate the control signals for its internal multiplexers. These counter signals are generated by a global counter unit and then propagated through the entire PA, delayed by an appropriate number of registers. The automatic generation of a global counter and control unit has been done as proposed in [17]. The number of registers is λ_{K_1} for the registers between two subsequent processor rows, and λ_{K_2} between two subsequent processor columns.

Processor Element. The internal structure of a processor is depicted in the zoomed view in Fig. 4. The boxes labeled D_{a0}, D_{u0}, D_{y0} contain the respective number of delay registers for the intra-processor (i.e., within one LS tile) data dependencies. The multiplexers select if the processor should compute with the internally stored values or read data from another processor or from outside the array. Depending on the processor's

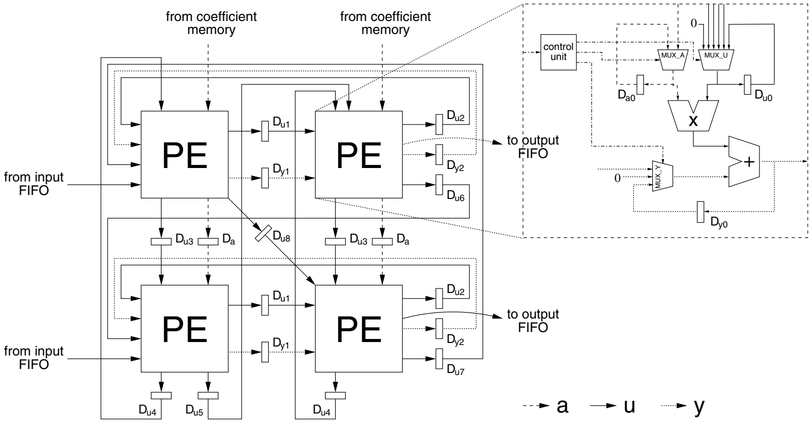


Fig. 4. Structure of the PA. Note that the length of the delay registers on the wrap around paths (e.g. $D_{u7} = 14$) might be large in that case external memory is used for corresponding data storage

position in the array, the MUX_U multiplexer has a different number of inputs. The control unit gets the iteration counter signals j_1, j_2, l_1, and l_2 and decodes them into select signals for the multiplexers. For the border PEs at $k_1 = 0$, $k_2 = 0$, and $k_2 = K_2 - 1$, this control unit also generates the control signals for the input/output FIFOs and the coefficient memories, respectively. To increase the clock frequency, the processors might be pipelined by using a pipelined multiplier and an additional pipeline register between the multiplier and the adder. Even with full pipelining, the multipliers are still the slowest component of the whole design.

I/O Model. With partial localization enabled and a sufficient number of processors available, the PA is able to process more than one sample per clock cycle. But in this case of course, the filter I/O ports must allow this higher sample rate. For this reason, the I/O ports are decoupled from the filter clock and operate at a different clock frequency. This is no problem because modern FPGAs offer several clock domains. Generally, the MAC unit tends to limit the overall clock frequency of a design. For example, on a Xilinx Virtex FPGA (xcv1000-4-bg560), using 16 bit input data and coefficients as well as a pipelined MAC unit, the maximum frequency of the PA can work at is about 65 MHz, while the maximum frequency for the I/O components is around 100 MHz. In this example, the maximum filter throughput would be $100MHz/65MHz = 1.54$. Asynchronous I/O FIFOs provide the interface between the two clock domains. The input FIFOs are filled with one sample per I/O clock cycle, while the PA reads the samples with the filter clock frequency. The data flow through the output FIFOs is just the other way round. The order in which the input samples are stored in the input FIFOs and the results are read from the output FIFOs is controlled by a global I/O control unit. Because more than one processor may read coefficients at the same time, each processor column gets its own, independent coefficient memory, which holds only those coefficients that are required by the corresponding subset of processors.

4 Automatic Filter Generation Tool

This section describes our C++ tool *Firgen*. Its purpose is to automate the steps partitioning, scheduling and VHDL generation for the synthesis of FIR filters onto an FPGA.

Automatic Partitioning. The first step is to choose an optimal partitioning scheme that meets the user's requirements with respect to number of filter taps, latency, throughput, clock frequency and costs. In order to find the optimal partitioning scheme, Firgen needs to know the effects of tile sizes and scheduling on resource usage and speed. This knowledge was obtained from a number of synthesis runs in order to characterize the building components (multiplier, register, etc.). Because these values are partly FPGA-dependent, they are stored in a configuration file and thus may be easily adapted for other devices. Also taking into account the theoretical facts from Section 2, Firgen looks for Pareto-optimal solutions that match the requested constraints. If they cannot be met, Firgen takes the solution closest to the user's preferences.

Scheduling. After the partitioning scheme is selected, the next step is to find an optimal scheduling. When determining the scheduling vector, Firgen takes into account the user specified latency and throughput constraints. After choosing the scheduling vector, the number of delay registers for each data dependency can be calculated. For the data dependency vector d and the scheduling vector λ, the number of resulting delay registers is $n = \lambda \cdot d$.

Optimization. During the filter generation, Firgen may perform the following optimizations:

- Selection of the optimal number of pipeline registers for the MAC units in order to maximize the clock frequency.
- Replace long shift registers that don't contain valid data at every stage by FIFOs. This reduces the number of required FPGA slices [5].

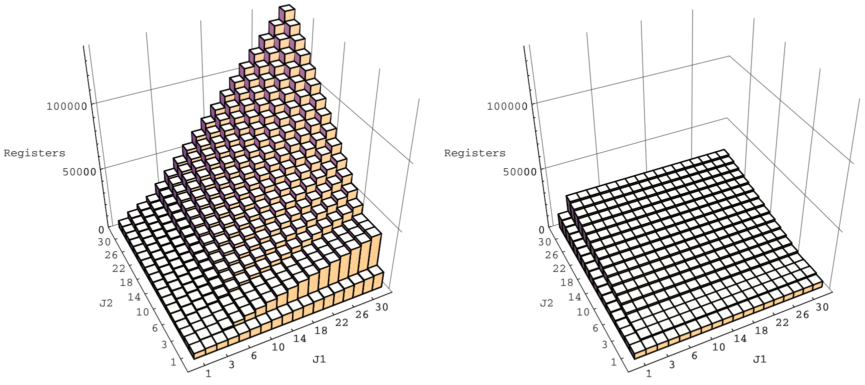
VHDL Generation. After all necessary parameters are determined by Firgen, the last step is to actually generate the VHDL code. Firgen generates an example component instantiation that the user can include in his design, and prints all necessary information, like the required ratio of I/O clock to filter clock frequency and the I/O latency. A VHDL test bench to verify the filter implementation may also be created.

5 Results and Comparison

In this section, we will explore the influence of the various partitioning parameters on resource usage and speed of our design. We also compare our design to FIR filters generated by *Xilinx Coregen*. All synthesis results were obtained from *Xilinx ISE 6.3i* on a *Xilinx Virtex* FPGA (xcv1000-4-bg560).

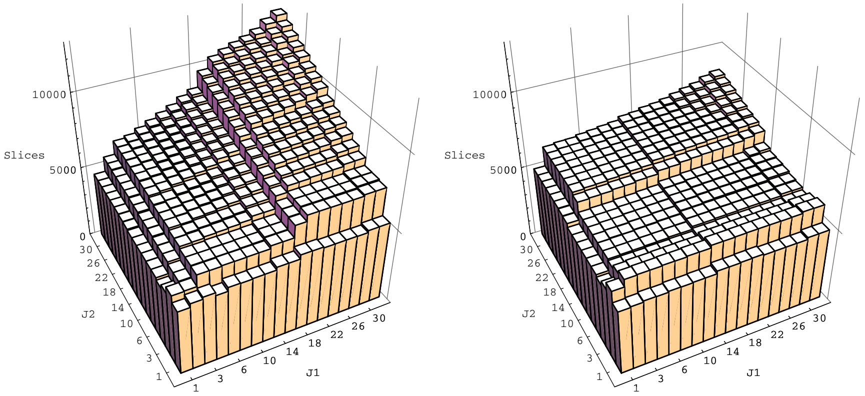
Fig. 5 (a) and (b) show the theoretical total number of 1 bit registers for different LS tile sizes (J_1, J_2) and the two possible LS scheduling directions. The parameters $K_1 = 4$, $K_2 = 4$ are fixed, resulting in a 4×4 PA, with fully pipelined MAC units, 16 bit input data, 16 bit coefficients and 40 bit filter output. The discrepancy between the theoretical number of registers and the actual FPGA resource usage as shown in Fig. 6 (a) and (b) has one major reason: Shift registers can be mapped to LUTs in a very efficient manner. One LUT can contain up to 16 bit of one shift register, therefore the slice count is proportional not to the number of registers n, but to $\lceil n/16 \rceil$, leading to steps in the diagram. The same is true for the input and output FIFOs and the coefficient memories, which also grow with greater tile sizes. The conclusions that can be drawn from the results are, (a) smaller tile sizes are generally preferable, (b) row-major scheduling leads to a significant saving of resources and considerably smaller latencies.

Xilinx Coregen offers also the possibility to generate classical MAC based filters (MAC FIR). We used Coregen to generate a filter with 64 taps, 16 bit signed input



(a) (b)

Fig. 5. Resource usage depending on LSGP tile sizes and LSGP scheduling direction, for an array with 4 × 4 processors, and the number of taps being $N = 4J_2$ (a) Total number of 1 bit registers for column-major LSGP scheduling. (b) Total number of 1 bit registers for row-major LSGP scheduling



(a) (b)

Fig. 6. Resource usage depending on LSGP tile sizes and LSGP scheduling direction, for an array with 4 × 4 processors, and the number of taps being $N = 4J_2$ (a) Number of FPGA slices for column-major LSGP scheduling, (b) Number of FPGA slices for row-major LSGP scheduling

data, 16 bit signed coefficients and 38 bit filter output. The filter throughput was set to 12.5% (0.125 samples per cycle) for all cases. Special features offered by Coregen implementations, exploiting coefficient symmetry, use of block RAM, that are not yet supported by Firgen were disabled to get comparable results. In Table 1 the results are shown. Note that the clock frequencies shown in the table are estimated by the synthesis tool, and obviously do not differ between the 'min area' and 'max speed' configurations of MAC FIR. Even after place-and-route, the resulting improvement of 'max speed' is only 0.7 MHz. So the differences of those two cases are unclear. In the current state,

Table 1. Resource usage and speed of Firgen, Xilinx Coregen MAC FIR generated FIR filters with 64 taps and a throughput of 12.5%

Tool	Configuration	Slices	Clock	Latency
Firgen (optimized for slices)	2 × 4 PEs	2271	64.107 MHz	68
Firgen (optimized for latency)	1 × 8 PEs	2744	61.263 MHz	20
Coregen MAC FIR	min area	2289	55.654 MHz	74
Coregen MAC FIR	max speed	2303	55.654 MHz	75

Firgen is as good as MAC FIR with respect to costs, and even better in terms of clock speed and latency. With partial localization enabled, Firgen has the advantage that it allows filter throughput greater than 100% with drastically reduced latencies but usually also for a higher prize.

6 Conclusions and Future Work

In this paper we used new transformations co-partitioning and partial localization for the automated generation of 2-d PAs for FIR filters. Considerable gains are obtained in throughput due to the mentioned transformations. The future work entails inclusion of efficient pipelined parallel multipliers as truncated multipliers [8] to achieve further increase in clock speeds. Also, Firgen can be easily adapted to FPGA's features such as block RAM and embedded multipliers. Furthermore, the Firgen tool is to be extended to handle standard VLSI signal processing algorithms as IIR filter, motion estimation, discrete wavelet transform etc.

References

1. Kienhuis, B., Rijpkema, E., Deprettere, E.: Compaan: Deriving process networks from matlab for embedded signal processing architectures. In: Proc. Int. Workshop Hardware/Software Co-Design, San Diego, U.S.A. (2000)
2. Schreiber, R., Aditya, S., Rau, B., Kathail, V., Mahlke, S., Abraham, S., Snider, G.: High-level synthesis of nonprogrammable hardware accelerators. Technical Report HPL-2000-31, Hewlett-Packard Laboratories, Palo Alto (2000)
3. Kathail, V., Aditya, S., Schreiber, R., Rau, B.R., Cronquist, D.C., Sivaraman, M.: PICO: Automatically designing custom computers. Computer **35** (2002) 39–47
4. Synfora, Inc.: (www.synfora.com)
5. Bednara, M., Teich, J.: Automatic synthesis of FPGA processor arrays from loop algorithms. The Journal of Supercomputing **26** (2003) 149–165
6. PARO Design System Project: (www12.informatik.uni-erlangen.de/research/paro)
7. CELOXICA, Handel-C: (www.celoxica.com)
8. Walters, E.G., Glossner, J., Schulte, M.J.: Automatic VHDL model generation of parameterized FIR filters. In: Proc. Int. Samos Workshop Systems, Architectures, Modeling, and Simulation. (2002)
9. Xilinx, Inc.: CORE Generator Guide, San Jose, CA, U.S.A. (2004)
10. Altera Corporation: FIR Compiler – MegaCore Function User Guide 3.2.0, San Jose, CA, U.S.A. (2004)
11. Guillou, A.C., Quinton, P., Risset, T., Massicotte, D.: Automatic design of VLSI pipelined LMS architectures. In: Proc. Int. Conf. Par. Comput. Electrical Eng., Quebec, Canada (2000) 144–149

12. Eckhardt, U., Merker, R.: Hierarchical algorithm partitioning at system level for an improved utilization of memory structures. IEEE T. CAD Integrated Circuits Syst. **18** (1999) 14–24
13. Teich, J., Thiele, L.: Exact partitioning of affine dependence algorithms. In Deprettere, E., Teich, J., Vassiliadis, S., eds.: Embedded Processor Design Challenges. Volume 2268 of Lecture Notes in Computer Science (LNCS). (2002) 135–153
14. Wolfe, M.: High Performance Compilers for Parallel Computing. Addison-Wesley Inc. (1996)
15. Hannig, F., Dutta, H., Teich, J.: Regular mapping for coarse-grained reconfigurable architectures. In: Proc. IEEE Int. Conf. Acoustics, Speech, Signal Process., Montréal, Quebec, Canada (2004) 57–60
16. Hannig, F., Teich, J.: Design space exploration for massively parallel processor arrays. In: Proc. Int. Conf. Par. Comput. Technologies, Novosibirsk, Russia (2001) 51–65
17. Dutta, H.: Mapping of Hierarchically Partitioned Regular Algorithms onto Processor Arrays. Master's thesis, University of Erlangen-Nuremberg (2004)

Two-Dimensional Fast Cosine Transform for Vector-STA Architectures

J.P. Robelly, A. Lehmann, and G. Fettweis

Vodafone Chair for Mobile Communications Systems,
Technische Universität Dresden,
01062 Dresden, Germany
robelly@ifn.et.tu-dresden.de

Abstract. A vector algorithm for computing the two-dimensional Discrete Cosine Transform (2D-VDCT) is presented. The formulation of 2D-VDCT is stated under the framework provided by elements of multilinear algebra. This algebraic framework provides not only a formalism for describing the 2D-VDCT, but it also enables the derivation by pure algebraic manipulations of an algorithm that is well suited to be implemented in SIMD-vector signal processors with a scalable level of parallelism. The 2D-VDCT algorithm can be implemented in a matrix oriented language and a suitable compiler generates code for our family of STA (Synchronous Transfer Architecture) vector architectures with different amounts of SIMD-parallelism. We show in this paper how important speedup factors are achieved by this methodology.

1 Introduction

The two-dimensional DCT plays a paramount role in video and image compression techniques. Over the years many fast algorithms have been proposed for the computation of the DCT. Most of the publications related to implementation issues of the DCT concentrate on VLSI implementations. We address in this paper the implementation of a fast algorithm for the DCT into our family of STA processor cores featuring SIMD-vector parallelism.

Over the past three decades, we have experienced how the SIMD-vector computational model has made its way from classical supercomputers to real-time embedded applications. In fact, vector signal processors have emerged upon the promise of delivering flexibility and processing power for computing number crunching algorithms at reasonable levels of power consumption. In [1] we presented a novel micro-architecture for designing and implementing low-power, high-performance DSPs cores. We call this architectural template Synchronous Transfer Architecture (STA). Moreover, in [2] we presented a hardware design methodology that enables the rapid silicon implementation of SIMD-vector processors with different levels of parallelism based on our STA architectural template.

The fast computation of signal transformations like the DCT is based on iterative divide-and-conquer algorithms: The transformation matrix is expressed as a function of smaller transformation matrices. Thus, the original computation that operates on vector

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 62–71, 2005.

spaces of a high dimensionality is reduced to the computation of smaller transformation matrices that operate on smaller vector spaces. The iterative formulation of the original transformation matrix is achieved by adequate permutation of the input samples. Elements of multilinear algebra are especially suitable for the description of this sort of algorithm. On the one hand, the rich framework offered by multilinear algebra allows for expressing the recursive nature of divide-and-conquer algorithms. On the other hand, it also enables the manipulation and derivation of new algorithms by exploiting pure algebraic properties. Especially interesting are those algebraic manipulations that reveal the vector operations of the algorithm, since they lead to formulations of algorithms that process data in vector fashion. These ideas are discussed in detail in [3], and they encouraged many researchers to publish a series of papers. Most of these papers address the derivation of vector algorithms for the classical example of the Fast Fourier Transform (FFT). Especially interesting is the work by Franchetti [4], where an algorithm for the vector computation of the FFT is presented.

In this paper we present the design of a vector algorithm for the computation of the two-dimensional DCT based on the framework of multilinear algebra. Once a suitable algorithm is designed, we implement it in a matrix oriented language like Matlab. Such a language allows for expressing vector algorithms described in the notation of multilinear algebra. A suitable compiler can recognize these operators and generate a sequence of vector machine instructions for our family of STA DSP cores. We show that important speedup factors are achieved by this methodology. The remainder of this paper is as follows. In section 2 we present our STA architectural template. In section 3 we introduce some elements of multilinear algebra. In section 4 we use this algebraic framework for the derivation of the 2D-VDCT algorithm. In section 5 we introduce our compiler infrastructure and the results obtained from the automatic code generation. Finally, in Section 6 we present our conclusions.

2 Synchronous Transfer Architecture STA

Our DSP architectural concept is based on basic modules of the form shown in Fig. 1. Such a basic module has an arbitrary number of input and outputs ports. In our concept the output of each module is a register. Input and output ports can deal with a certain data type, e.g. bool, 16-bit integer, 32-bit floating point, vectors of 16-bit integer, vectors of 32-bit floating point, etc. Basic modules implement some functionality. In our architecture concept, a system is built up from basic modules. Thus, ports of the same data type are connected with each other through an interconnection network formed by multiplexers.

Both the functionality of basic modules and the input multiplexers are explicitly controlled by processor instructions. At each cycle the instruction configures the multiplexing network and the functionality of the basic modules. Thus, the whole system forms a synchronous network, which at each clock cycle consumes and produces some data. The produced data will in turn be consumed by other basic modules in the next cycle. Due to the synchronous transfer of data between basic modules we have named the architecture STA. In our concept, basic modules can be highly optimized data paths or memory blocks. Memory blocks can be either register files or memories.

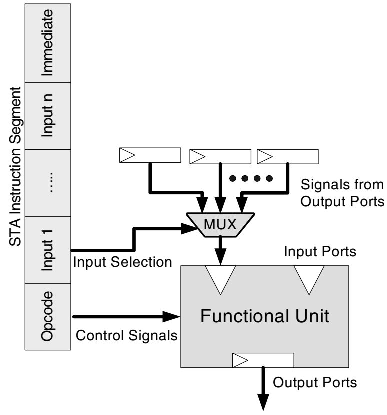


Fig. 1. STA Architectural Template

The STA architecture offers a high degree of data reusability: Data produced in the current cycle is directly routed to other processing units in the next cycles without going through the register file or memory. This not only speeds up computations but it lowers power consumption and register file pressure. The STA architecture also supports data and instruction level parallelism. In fact, SIMD-vector data parallelism is supported by letting input ports, output ports and data paths to deal with vector data types. Instruction level parallelism is supported, since at each cycle a wide instruction controls each basic module and the multiplexing network. This poses two problems. On the one hand, for large STA systems the multiplexing interconnection network becomes a critical part of the design. On the other hand, a wide instruction memory is needed. The complexity of the interconnection network can be alleviated by reducing the number of connections between ports. An obvious strategy for this is to determine those connections which allow for reusing program variables that present a high data locality. This is a viable approach, since applications are known at design time of the processor and thus the interconnection network can be customized. For those connections which are not frequently reused, a connection with the register file suffices. To alleviate instruction memory footprint we are applying code compression techniques similar to [5].

The simplicity and modularity of the STA concept enables the automatic generation of RTL and simulation models of processor cores from a machine description [2]. This allows for generating processor cores with different characteristics, e.g. size of register file, memory capacity, interconnection network, functional units, data types and amount of SIMD-vector parallelism. Despite the architecture simplicity of STA, it can handle many applications of considerable complexity. For example a generic STA processor [1] with 8 data paths running at 212 Mhz. executes a 256 complex FFT in $8\mu s$. The automatic code generation of processor cores imposes a new challenge: How to design and implement algorithms that can be reused for a scalable level of SIMD-vector paral-

[1] STA processor core furnished only with multipliers and adders. No customized functional units for FFT acceleration (complex arithmetic, bit reverse or butterfly units) are available.

lelism? In order to face this problem some sort of abstraction is needed. From the code generation perspective the adequate abstraction is achieved by applying compiler techniques that can be reconfigured according to the target processor. From the algorithm design perspective the adequate abstraction is achieved by using an algebraic framework that resembles the features of the processor.

3 The Algebraic Characterization of SIMD Parallelism

Let us assume the computation $\mathbf{y} = A\mathbf{x}$, where $\mathbf{y}, \mathbf{x}$ are some vectors whose components are scalars. The transformation matrix A defines an algorithm which in order to be computed in a serial computer requires a certain number of machine cycles c. Now, let us consider that the same transformation is to be computed in a vector processor with a level of parallelism ν. It is a fact that in the same number of considered machine cycles c the transformation described by the matrix A can be computed for ν different vectors. This can be expressed as

$$\tilde{\mathbf{y}} = (A \otimes I_\nu)\,\tilde{\mathbf{x}}, \tag{1}$$

where $\otimes$ is the Kronecker product, I_ν is a $\nu \times \nu$ identity matrix, and $\tilde{\mathbf{x}}, \tilde{\mathbf{y}}$ can be regarded as vectors with ν components and each component is itself a vector of the same dimensionality as $\mathbf{x}$ and $\mathbf{y}$ respectively. Equation (1) captures the SIMD computational model and thus it is called a Kronecker SIMD-Vector factor [3]. More generally, the expression

$$(I_b \otimes A \otimes I_\nu \otimes I_c) \tag{2}$$

can be fully vectorized and thus it can be efficiently implemented in a vector processor with a level of parallelism ν. As we can observe the Kronecker product, which is an operator from multilinear algebra, plays an important role in the description of algorithms for SIMD-vector processors. Consider the $m_1 \times n_1$ matrix A with entries $\left[a_{j,k}\right]$ for $j = 1, 2, \ldots, m_1$ and $k = 1, 2, \ldots, n_1$, and the $m_2 \times n_2$ matrix B, then the $m_1 m_2 \times n_1 n_2$ matrix C that results from the Kronecker product of A and B is defined as

$$C = A \otimes B = \begin{bmatrix} a_{1,1}B & a_{1,2}B & \ldots & a_{1,n_1}B \\ a_{2,1}B & a_{2,2}B & \ldots & a_{1,2}B \\ \vdots & \vdots & & \vdots \\ a_{m_1,1}B & a_{m_1,2}B & \ldots & a_{m_1,n_1}B \end{bmatrix}. \tag{3}$$

Another important concept from multilinear algebra is the direct sum. The direct sum of n arbitrary matrices is defined as

$$\bigoplus_{i=0}^{n-1} A_i = \begin{bmatrix} A_0 & 0 & \cdots & 0 \\ 0 & A_1 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & A_{n-1} \end{bmatrix}. \tag{4}$$

To some extent, multilinear algebra can be regarded as the branch of algebra that deals with the construction of vector spaces of a higher dimensionality from a set of primitive vector spaces. The Kronecker product and the direct sum are the two fundamental concepts of algebra that enable this construction of vector spaces of a higher dimensionality.

Especially intriguing is the connection between Kronecker products and shuffle algebra. Davio, in his classical paper [6] established the connection between the Kronecker product of matrices and stride permutations. In his paper he proved the important so called commutation theorem of Kronecker products

$$(A \otimes B) = P^N_M (B \otimes A) P^N_L, \qquad N = ML, \tag{5}$$

where A, B are $M \times M$, $L \times L$ matrices and P^N_L is an N-point stride by L permutation. Additional useful identities for manipulating stride permutations are

$$P^{MLK}_K = \left(P^{MK}_K \otimes I_L\right)\left(I_M \otimes P^{LK}_K\right), \tag{6}$$

$$P^{MLK}_{ML} = \left(I_M \otimes P^{LK}_L\right)\left(P^{MK}_M \otimes I_L\right), \tag{7}$$

$$I_{ML} = I_M \otimes I_L. \tag{8}$$

Kronecker products present a series of other interesting algebraic properties. For example, for the above introduced square matrices A and B we can write

$$A \otimes B = (A \otimes I_L)(I_M \otimes B). \tag{9}$$

4 2D-VDCT

In this section we introduce the mathematical derivation of the two-dimensional DCT adapted to SIMD-vector processing. At first, we introduce the one-dimensional DCT algorithm and later we extend the algorithm to the two-dimensional case.

4.1 1-D Fast Cosine Transform Algorithm

In [7], Cvetković developed an algorithm for the one-dimensional DCT based on the algorithm derived by Hou [8]. The fast algorithm described in this paper presents a factorization of the N-point Type-II 1D-DCT as a product of sparse matrices. This factorization can be expressed in terms of elements of multilinear algebra as follows

$$\begin{aligned}\mathrm{DCT}_N = Q_N &\left[\prod_{k=0}^{q-2}\left(I_{2^k} \otimes A_{\frac{N}{2^k}}\right)\right]\left[\prod_{k=1}^{q}\left(I_{\frac{N}{2^k}} \otimes B_{2^k}\right)\right. \\ &\left.\cdot\left(I_{\frac{N}{2^k}} \otimes \mathrm{DFT}_2 \otimes I_{2^{k-1}}\right)\right] R_N, \end{aligned} \tag{10}$$

where $q = \mathrm{ld}(N)$. The bit-reversal matrix can be defined in terms of stride permutations as

$$Q_N = \prod_{k=0}^{q-2}\left(I_{2^k} \otimes P^{2^{q-k}}_{2^{q-k-1}}\right).$$

A_N is a sparse matrix involved in the computation and is defined as

$$A_N = \left(I_{\frac{N}{2}} \oplus K_{\frac{N}{2}}\right), \tag{11}$$

where I_N is an $N \times N$ identity matrix and $K_N = Q_N L_N Q_N$ for the $N \times N$ matrix

$$L_N = \begin{bmatrix} 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \ddots & \vdots \\ 0 & 0 & 1 & \ddots & 0 \\ \vdots & & \ddots & \ddots & 1 \\ 0 & & \cdots & 0 & 1 \end{bmatrix}. \tag{12}$$

Other sparse matrix involved in the computation is B_N, which is defined as

$$B_N = \left(I_{\frac{N}{2}} \oplus C_{\frac{N}{2}}\right), \tag{13}$$

where

$$C_{\frac{N}{2}} = \text{diag}\left[\frac{1}{2\cos(\phi_m)}\right], \quad m = 0, 1, \ldots, N/2 - 1 \tag{14}$$

and

$$\phi_m = \frac{2\pi(m + 1/4)}{N}.$$

The matrix DFT_2 denotes the 2-point Discrete Fourier Transform (DFT) matrix and it is given by

$$\text{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{15}$$

Finally, the input permutation matrix R_N is defined as

$$R_N = \left(I_{\frac{N}{2}} \oplus \bar{I}_{\frac{N}{2}}\right) P_2^N,$$

where $\bar{I}_{N/2}$ is the mirrored $N/2 \times N/2$ identity matrix. For example, $\bar{I}_{\frac{N}{2}}$ for $N = 8$ is given by

$$\bar{I}_{\frac{8}{2}} = \begin{bmatrix} 0\,0\,0\,1 \\ 0\,0\,1\,0 \\ 0\,1\,0\,0 \\ 1\,0\,0\,0 \end{bmatrix}.$$

4.2 Derivation of the 2-D VDCT Algorithm

Pratt [9] points out that separable bilinear transformations can be expressed as two linear transformations: one operating over the rows and one operating over the columns. Since the 2D-DCT can be regarded as such a bilinear transformation we can write for the transformation of an $N \times N$ matrix X the following

$$(\text{DCT}_N)\, X\, (\text{DCT}_N)^T. \tag{16}$$

The computation of the 2D-DCT in the matrix space as in equation (16) is isomorph to a computation of the algorithm that operates onto a vector space. The vector space is constructed by stacking the rows of the matrix X. For this case the computation of the 2D-DCT becomes

$$(\mathrm{DCT}_N \otimes \mathrm{DCT}_N) \cdot \mathrm{Vec}\,(X), \tag{17}$$

where $\mathrm{Vec}\,(\cdot)$ represents the vectorization of a matrix in row-major order. Thus, the two-dimensional algorithm can be derived from the one-dimensional matrix form using the identity

$$\mathrm{DCT}_{N\times N} = \mathrm{DCT}_N \otimes \mathrm{DCT}_N. \tag{18}$$

This 2-D transformation matrix has to be applied to a vector $\mathbf{x}$ of N^2 elements. The vector $\mathbf{x}$ is obtained from the row-major ordering of the $N \times N$ input data array $X(n,n)$ and thus it is defined as

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_0^T \\ \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_{N-1}^T \end{bmatrix} \qquad \text{for} \quad \mathbf{x}_n = [X(n,1)\ldots,X(n,N-1)].$$

Using (9), we can write for equation (18) the following

$$\mathrm{DCT}_{N\times N} = (\mathrm{DCT}_N \otimes I_N)(I_N \otimes \mathrm{DCT}_N). \tag{19}$$

In order to derive an algorithm that process data in vector fashion, expressions of the form given by (1) are required. By using identity (5), (19) can be expressed as

$$DCT_{N\times N} = (DCT_N \otimes I_N)P_N^{N\cdot N}(DCT_N \otimes I_N)P_N^{N\cdot N}, \tag{20}$$

where the maximal SIMD parallelism of $\nu_{max} = N$ is obtained. The expression $P_N^{N\cdot N}$ denotes a transposition matrix that cannot be efficiently mapped to SIMD processors, since it does not match the form given by (2). For this reason, a further decomposition is required to obtain a simpler structure for vector processors. Using identities (6) and (7) yields to a formulation adapted to a level of parallelism ν. We obtain for the transposition

$$P_N^{N^2} = \left(I_{\frac{N}{\nu}} \otimes P_\nu^N \otimes I_\nu\right)\left(I_{\frac{N^2}{\nu^2}} \otimes P_\nu^{\nu^2}\right)\left(P_{\frac{N}{\nu}}^{\frac{N^2}{\nu}} \otimes I_\nu\right). \tag{21}$$

The term $(\mathrm{DCT}_N \otimes I_N)$ adapted to a SIMD vector length ν, using the selected 1D-algorithm (10) and identity (8) is given by

$$\begin{aligned}
(\mathrm{DCT}_N \otimes I_N) = {} & \prod_{k=0}^{q-2}\left(I_{2^k} \otimes P_{2^{q-k-1}}^{2^{q-k}} \otimes I_\nu \otimes I_{\frac{N}{\nu}}\right) \\
& \cdot \left[\prod_{k=0}^{q-2}(I_{2^k} \otimes A_{\frac{N}{2^k}} \otimes I_\nu \otimes I_{\frac{N}{\nu}})\right]\left[\prod_{k=1}^{q}\left(I_{\frac{N}{2^k}} \otimes B_{2^k} \otimes I_\nu \otimes I_{\frac{N}{\nu}}\right)\right. \\
& \cdot \left.\left(I_{\frac{N}{2^k}} \otimes \mathrm{DFT}_2 \otimes I_\nu \otimes I_{2(q-p+k-1)}\right)\right]\left(R_N \otimes I_\nu \otimes I_{\frac{N}{\nu}}\right) \\
& q = \mathrm{ld}(N),\, p = \mathrm{ld}(\nu),\ p \leq q.
\end{aligned} \tag{22}$$

Applying (21) and (22) into (20), we obtain a fast 2-D VDCT algorithm for SIMD-vector processing. The derived algorithm is completely parameterizable by the transformation size N of the source data array and the available level of parallelism ν of the used processor. Nearly all terms of the algorithm are vector computations matching expression (2) excepting for $\left(I_{N^2/\nu^2} \otimes P_{\nu}^{\nu^2}\right)$.

5 Implementation and Results

The algorithm derived above can be directly implemented in a matrix-oriented languages like Matlab. In [2], we presented a compiler infrastructure for the automatic code generation for our STA SIMD-vector cores. In Fig. 2, we can observe a block diagram of the compiler. The compiler stage known as Kronecker center part features the pattern matching of expressions embedded in the Matlab code, which have the form of equation (2). The Matlab code is programmed using functions that compute the different operators of multilinear algebra and stride permutations introduced in section 3. This compiler stage also generates a high level intermediate representation of vector instructions that will be processed by further stages of the compiler. This high level intermediate representation of the program resembles a linear instruction list.

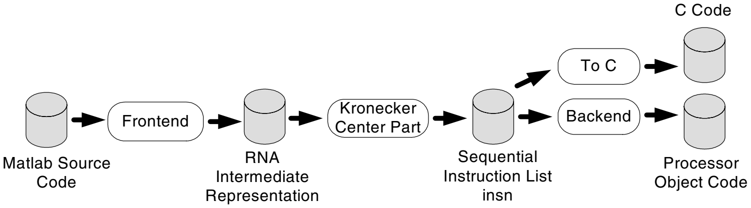


Fig. 2. Block Diagram of the Compiler Infrastructure for Automatic Code Generation

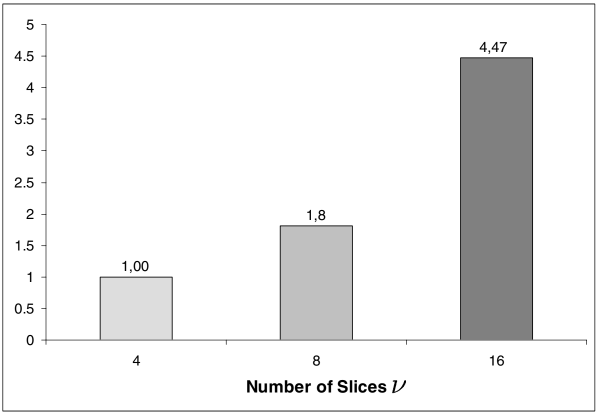


Fig. 3. Speed-up factors for $\mathrm{DCT}_{16\times16}$

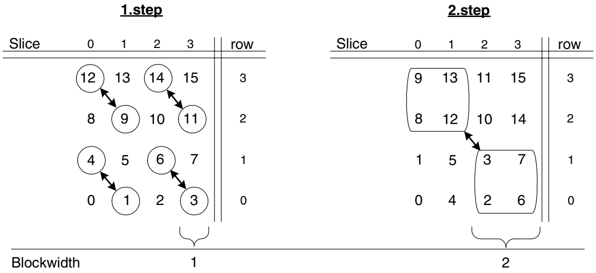


Fig. 4. Data transfers for computing $P_{\nu}^{\nu^2}$ for $\nu = 4$

The algorithm presented in section 4 was programmed in Matlab and object code was generated for our DSP processor cores with different levels of parallelism ν. For N=16 speedup results for a raising number of data paths ν are presented in Fig.3. The speedup is computed taking as a reference the execution time required to compute the algorithm into a DSP processor with a level of parallelism $\nu = 4$. As we can observe from the diagram, we can expect to achieve a speedup factor of 1.8 if we implement the algorithm into a machine with $\nu = 8$ data paths. A more important gain on the speedup factor is achieved if the level of parallelism is $\nu = 16$. For this case some expensive permutations of (21) are cancelled out.

In order to reduce the impact of the unique non-vector term of our algorithm, namely $P_{\nu}^{\nu^2}$, it is important to find an efficient implementation. A proposed approach is described in Fig. 4 for $\nu = 4$. Our processor architecture is furnished with an interconnection network that supports the required data transfers between data paths in $i = \mathrm{ld}\nu$ steps.

6 Conclusions

The 2D-VDCT algorithm has been derived by pure mathematical means using concepts of multilinear algebra. Since almost all terms involved in the computation of 2D-VDCT process data in vector fashion, the algorithm is especially efficient for SIMD-vector processors. The algorithm is completely parameterized for a certain transformation size N and the available level of SIMD parallelism ν. We have presented a compiler infrastructure that can generate code for our family of STA DSP cores from a Matlab program. The computation of the algorithm in a processor with $\nu = 16$ data paths can be up to a factor of 4,47 times faster than the implementation of the algorithm into a processor with $\nu = 4$ data paths. We believe that the methodology presented in this paper offers an interesting approach to close the gap between fast algorithms for signal processing, compiler technology and processor architecture.

References

1. Cichon, G., Robelly, P., Seidel, H., Bronzel, M., Fettweis, G.: Synchronous transfer architecture (STA). In Vassiliadis, S., ed.: Lecture Notes on Computer Science. Springer-Verlag, Berlin, Germany (2004) 343–352

2. Robelly, P., Cichon, G., Seidel, H., Fettweis, G.: A hw/sw design methodology for embedded simd vector signal processors. International Journal of Embedded Systems IJES (2005)
3. Tolimieri, R., An, M., Lu, C.: Algorithms for Discrete Fourier Transform and Convolution. Springer Verlag, Berlin, Germany (1997)
4. Franchetti, F., Pueschel, M.: Short vector code generation for discrete fourier transform. In: In Proc. International Parallel and Distributed Processing Symposium (IPDPS). (2003) 58–67
5. Weiss, M., Fettweis, G.: Dynamic codewidth reduction for vliw instruction set architectures in digital signal processors. In: In Proc. of the 3rd. Int. Workshop in Signal and Image Processing IWSIP'96. (1996) 571–520
6. Davio, M.: Kronecker products and shuffle algebra. IEEE Trans. on Computers **C-30** (1981) 116–125
7. Cvetković, Z., Popović, M.V.: New fast recursive algorithms for the computation of the discrete cosine and sine transforms. IEEE Trans. on Signal Processing **40** (1992) 2083–2086
8. Hou, H.: A fast recursive algorithm for computing the discrete cosine transform. IEEE Trans. on Acoustics, Speech and Signal Processing **35** (1987) 1455–1461
9. Pratt, W.: Digital Image Processing. John Wiley and Sons (1991)

Configurable Computing for High-Security/High-Performance Ambient Systems*

Guy Gogniat[1], Wayne Burleson[2], and Lilian Bossuet[1]

[1] Laboratory of Electronic and REal Time Systems (LESTER),
University of South Britanny (UBS), Lorient, France
{guy.gogniat, lilian.bossuet}@univ-ubs.fr

[2] Department of Electrical and Computer Engineering,
University of Massachusetts, Amherst, MA 01003-9284, USA
burleson@ecs.umass.edu

Abstract. This paper stresses why configurable computing is a promising target to guarantee the hardware security of ambient systems. Many works have focused on configurable computing to demonstrate its efficiency but as far as we know none have addressed the security issue from system to circuit levels. This paper recalls main hardware attacks before focusing on issues to build secure systems on configurable computing. Two complementary views are presented to provide a guide for security and main issues to make them a reality are discussed. As the security at the system and architecture levels is enforced by agility significant aspects related to that point are presented and illustrated through the AES algorithm. The goal of this paper is to make designers aware of that configurable computing is not just hardware accelerators for security primitives as most studies have focused on but a real solution to provide high-security/high-performance for the whole system.

1 Introduction

Configurable computing research area has been deeply studied these last ten years. Today its maturity is largely admitted and many works have demonstrated its efficiency. As a consequence, configurable computing is now widely used in embedded systems to provide system performances and flexibility. At the same time pervasive computing is becoming a reality which enables interconnecting systems in a huge network [1]. As all entities can communicate to exchange data the critical question of security is unavoidable. Privacy and confidentiality is a major issue for users [2]. At the hardware level configurable computing offers numerous interesting features to efficiently handle this point. However, since now most studies have focused on the use of configurable computing to speed up security primitives dealing with architectural optimizations whereas it is also mandatory to consider configurable computing at all levels from system to circuit. In this paper the problem of hardware security related to configurable computing

* This work is supported by the French DGA DSP/SREA under contract no. ERE 04 60 00 010.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 72–81, 2005.

is addressed. A quick review of hardware attacks is first presented in order to emphasize the main features a system must provide to be secure. Then configurable computing is analyzed to demonstrate its ability to address these features. Based on this information a guide for hardware security using configurable computing is proposed. Main issues to make this guide a reality are then discussed. Finally, as most of these concepts rely on the agility property provided by configurable computing a case study dealing with the AES security primitive is proposed to illustrate that point.

2 Hardware Attacks and Counter-Measures

Hardware Attacks. Two types of attacks are considered depending on the way the attack is performed: active or passive attack. Active attack which corresponds to an alteration of the normal device operation can be further refined into three subtypes. Irreversible attack is a physical attack and corresponds to chip destruction or modification for reverse-engineering. After this type of attack the device does not perform its initial computation or not at all. Reversible attack consists in punctually moving the device out of its specified operation modes so as to move it into a weak state or to gain information from a computation fault [3]. Reversible attack can be or not detectable. When detected the device can react in order not to leak any information (e.g., by erasing a private key). Non detectable attacks can be for example glitch attacks on clock or power. It is very difficult to detect these types of attacks as there is always a compromise between reliability and efficiency. When too sensitive the detection is not reliable since it can detect some normal variations that do not correspond to any attacks and when not enough sensitive the detection is neither reliable since it cannot detect some attacks. Detectable attack is for example black box attack, fault injection and power or temperature reduction. For a whole system they are difficult to handle. However one has to keep in mind that given enough time, resources and motivation an attacker can break any system [4]. Passive attacks enable to deduce secrets from the analysis of the correlation between the legal information output by the device and the side-channel information (i.e., current, power, electromagnetism) [3]. In that case the device computes normally and the attack is more sophisticated since it relies only on the statistical evolution of the peripheral information. Examples of passive attacks are timing, power and electromagnetic emission analysis [5].

Counter-Measures. So what conclusions from these attacks must be drawn in order to increase system security at the hardware level? To be safe a system should:

- Not provide any information (i.e., data leaks) in order to disable passive attacks. The system must be symptom-free [3].
- Be continuously aware of its state and notably of its vulnerability in order to react if necessary. The system must be security-aware.
- Analyze its states and its environments in order to detect any irregular activity. The system must embed distributed sensors and monitors to be activity-aware.
- Be agile in order to react rapidly to an attack or to anticipate an attack. Be agile to be a ble to update security mechanisms as long as attacks evolve. The system must provide agility.

- Be tamper resistant in order to resist to physical attacks. The system must be robust [6][7].

And at the same time, the system must provide high performance to run the applications. Throughput, latency, area, power, energy are examples of parameters that are mandatory to run actual applications. So where is the solution, what technology provides these characteristics?

3 Configurable Computing

Configurable computing presents several major advantages to deal with both security and performance compared to dedicated hardware components and processors. Some aspects are not specific to configurable computing but are more related to design at logic and circuit levels as for example symptom-free and robustness. But one major feature is required when dealing with security, adaptability (this term gathers the notions of awareness and agility) that is not provided by dedicated hardware components. Processors also provide adaptability through code update but they do not meet with the high performance requirements. Configurable computing provides many interesting features to be selected as a high-security/high-performance target. One key feature that underlies all others is the dynamic nature of configurable computing. Dynamic configuration enables to react and adapt rapidly in order to provide efficient architecture for performance and security. Irregular activities can be detected and the system can react objectively. This dynamism can be performed at run time or not, depending on the requirements. High performance with configurable computing has been deeply studied these last ten years. In the special case of security these studies have focused and still focus on high performance for security primitives (typically cryptography) [8][9]. However this vision of security is only a part of the challenge to be addressed to provide secure system since it deals mainly with architecture optimizations. In the following a vision of security that gathers all the issues from system to circuit levels is proposed. It is essential to enlarge today vision of security since configurable computing may not be a single part of the system but the whole system.

4 Secure Systems and Configurable Computing

When dealing with security it is important to determine what you want to protect and for how long you want to protect it. Furthermore defining the proper security boundary is critical for designing a flexible yet provably secure system [4]. In order to address these points it is essential to analyze what the different issues to provide secure configurable computing are. The very important idea is to classify and to define what the boundary of each part of a configurable system is and what design levels for security have to be targeted. Depending on the design and the security policy one or several security issues have to be considered. The designer has to be aware of these boundaries and design levels in order to guide his safe design building. In the following two complementary views are proposed in order to deal with security and configurable computing, the first one is related to system parts (system boundaries) and the second one to system security layers (security hierarchy).

4.1 Configurable Computing Security Space

The view of a system and of its different parts enables to highlight what the issues to build secure systems are. Three domains have to be addressed: Configurable Security Module, Secure Configurable System, and Configurable Design Security. Each domain focuses on a specific point and is detailed hereafter.

Configurable Security Module. A Configurable Security Module is a part of the whole system and performs some security primitives (e.g., cryptography, data filtering). A system generally embeds several Configurable Security Modules. Many works have focused to define efficient Configurable Security Modules dealing with very interesting and optimized architectures [8][9], however when dealing with agility it is also essential to define what the rules to switch or to update a module are. Thus, a security module controller is needed in order to manage the agility (i.e., flexibility) provided within the Configurable Security Module. Typical security module controller control tasks are related to configuration context to change or to adapt the functionality of the module [10].

Secure Configurable System. The Secure Configurable System domain deals with the security of the whole system to mainly perform intrusion prevention and detection. To build a Secure Configurable System three main points have to be considered: security awareness, activity awareness through distributed sensors and monitors, and agility. Security awareness is required in order to build a system that is aware of its state in order to anticipate and to detect possible attacks. Distributed sensors and monitors build the security network that enables the system to be aware of its activity [10]. Agility enables the system to react in order to modify its state to defeat an attack. Different levels of reaction are considered depending on the type of attack, reflex or global. Typical monitor control tasks are related to sensors and protocols analysis, monitor state exchange, reaction management, and monitor agility.

Configurable Design Security. A configurable computing module/system is defined through the configuration data since each hardware execution context is defined through a specific configuration. The configuration data represents the design of the module/system; it may contain private information (i.e., intellectual property) that needs to be protected from adversaries to prevent reverse-engineering. The design security is provided through cryptography and needs a dedicated Configurable Security Module that performs the cryptography primitives (i.e., authentication, encryption). When the configuration data is protected the Configurable Security Module enables to configure the system without leaking any information about the design it embeds [11].

Depending on the security policy one or all domains have to be considered. Right now at the hardware level most studies have focused on Configurable Security Module and Configurable Design Security however it is essential to deal with the system level architecture to enable the visions of ubiquitous computing [10]. It is important to keep in mind that building a secure system has some overhead costs, so defining the right security boundary is important to meet with design constraints and to provide power efficient system [12]. Configurable computing enables to provide security/performance trade-off dynamically which promotes dynamic evolution of the system to manage dynamic security policy.

4.2 Configurable Computing Security Hierarchy

The previous view was dealing with the different parts of a system which is important for the designer in order not to disregard any parts of the security barriers. Another view is important which is related to the different hierarchical levels of a design from system to circuit levels. As each level provides specific weaknesses specific mechanisms need to be defined in order to build a global secure system (i.e., defense in depth). Depending on the requirements several levels have to be considered however as previously mentioned it is important to clearly define what the security boundaries for a system to be protected are. In the following main issues dealing with each level are presented.

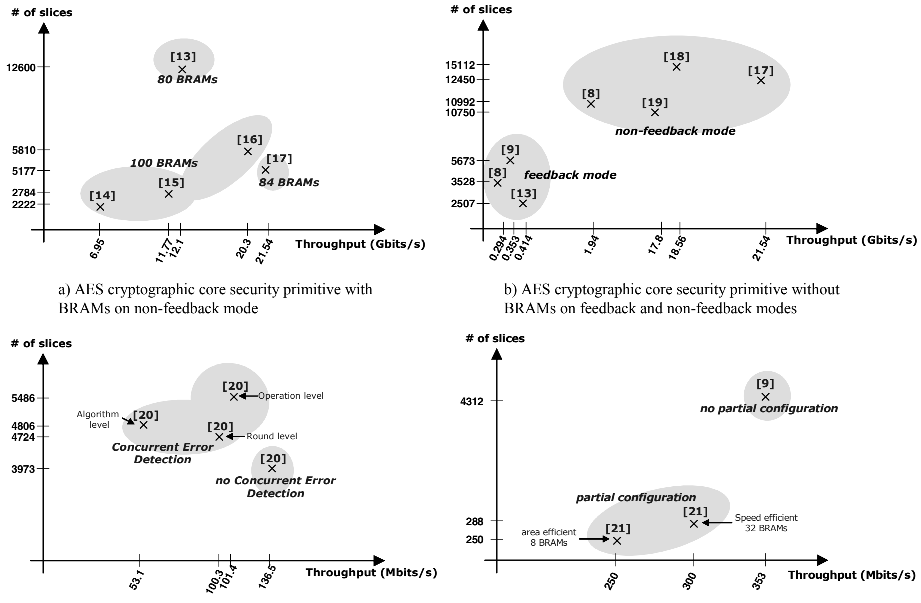
Secure System Level. At the system level configurable computing is seen as the global system (it corresponds to the Secure Configurable System in the previous view). At that level it is important to continuously monitor the activity of the system to detect irregular sequence of computation [10]. Another important feature is to keep the system as a moving target in order not to enable attackers to get a signature of the system or to identify some sensitive parts of the system [4]. This mobility should be provided for both system parts and monitors.

Secure Architecture Level. At the architectural level the architecture of a module is considered (it corresponds to the Configurable Security Module in the previous view). Critical modules are typically cryptography primitives. The architecture of these modules has to be flexible, efficient and fault tolerant. Another important feature for security is to provide symptom-free and security-aware algorithms and modules in order to disable side-channel attacks.

Secure Logic Level. At the logic level the design of gates is targeted. The main point that has to be considered is to provide symptom-free gates (e.g., balance the computation time, synchronize the inputs, leave no evidence of previous computation) [3]. Gates need to be fault tolerant in order to be reliable. Reliability has to be a major concern since fault injection can break security barriers.

Secure Circuit Level. At the circuit level the transistors and the physical process are considered. The goal is to strengthen the hardware physical shielding against for example RAM overwriting, optical induced fault, clock or power glitch attacks. An essential issue at that level is to define sensors that enable to prevent attacks by detecting them.

Configurable Computing Security Hierarchy is a complex structure and each level has its importance in order to provide a defense in depth. There are still many open problems to provide such a hierarchy from physical to system level concepts. The key idea



a) AES cryptographic core security primitive with BRAMs on non-feedback mode

b) AES cryptographic core security primitive without BRAMs on feedback and non-feedback modes

c) AES cryptographic core security primitive with and without concurrent error detection mechanisms on feedback mode

d) AES cryptographic core and key setup security primitive using or not partial configuration

Fig. 1. Agility design space for the AES security primitive: throughput/area/reliability trade-offs

is always to provide symptom-free and security-aware devices; the strength of configurable computing is its inherent agility using dynamic configuration. It enables to keep the system moving and to strengthen the security barriers when needed. Another essential issue when dealing with embedded systems is to provide the just right barrier and efficiency in order to keep alive the system functionality as long as possible (i.e., power-aware systems). Configurable computing provides this capability but the mechanisms to control this adaptability still need to be defined. As most of the concepts presented in the two previous views rely on agility, in the following a discussion dealing with that point is provided.

5 AES (Rijndael) Security Primitive Agility Case Study

To illustrate the concepts related to agility an analysis of a Configurable Security Module is proposed. The case study deals with the AES security primitive. This case study is based on published works dealing with configurable architecture. All the selected implementations have been performed on Xilinx Virtex FPGA which is a fine grain configurable architecture. For that architecture the configuration memory relies on a 1D configuration array. More precisely it is a column based configuration array, hence partial configuration can be performed only column by column. For security issues, this type of configuration memory does not provide full flexibility but still enables partial dynamic configuration to perform security scenarios. Figure 1 gathers all the different implementations and represents them in four charts; each chart corresponds to some specific parameters. Figure 1.a corresponds to the AES cryptographic core security primitive with BRAMs (i.e., embedded RAM) on non-feedback mode [13][14][15][16][17]. Thus for these studies key setup management is not considered. Concerning agility all the solutions are based on static and full configuration. The configuration is defined through predefined configuration data and performed using remote-configuration. The configuration time is on average tens of ms, since full configuration is performed. The security module controller is not addressed in these studies since the implementations are static. Figure 1.a highlights that various area/throughput trade-offs can be provided depending on the implementation. This is important to dynamically adapt the performance and to deal with security of the module. From the security side it enables the global system to behave as a moving target and from the performance side it allows to dynamically consider different throughputs depending on the actual application requirements. Figure 1.b corresponds to the AES cryptographic core security primitive without BRAMs on feedback [8][9][13] and non-feedback modes [8][17][18][19]. As previously key setup management is not considered. Solutions [8], [9] and [13] correspond to feedback mode while the others to non-feedback mode. Feedback solutions provide throughput on average hundreds of Mbits/s whereas non-feedback solutions are around tens of Gbits/s. Same remarks as previously can be done concerning agility characteristics; static, full and predefined configuration is considered. The goal in these studies is to promote high throughput while reducing area and dealing with a specific execution mode. However as explained in this paper dynamism and reliability have to be considered also.

Figure 1.c is interesting since it proposes different solutions that manage fault detection which guaranties reliability; essential feature for security. Fault detection can be performed at different levels of granularity from algorithm to operation level [20]. Performance/reliability trade-off is interesting since finer level of granularity enables reduced fault detection latency and then promotes fast reaction against an attack. But this efficiency is at the price of area overhead. No error detection leads to better performance, thus is it important to dynamically adapt the level of protection depending on the environment and the state of the system. Concerning agility static, full and predefined configuration is considered. Finally Fig. 1.d provides some interesting values since solutions dealing with dynamic configuration are proposed. In [9] full configuration with predefined configuration data is implemented whereas in [21] partial configuration with dynamic configuration data is carried out. In both cases remote-configuration is performed since the Configurable Security Module is seen as an agile hardware accelerator. Both solutions also deal with key setup management, in [9] it is performed within the module so that the architecture is generic and in [21] it is performed by the remote-processor which enables to provide key-specific architecture. In [21] the remote-processor implements the security module controller that computes the new configuration when new keys have to be taken into account by the cryptography core. This type of execution enables a large flexibility since the configuration data can be defined at runtime. However, in that case the computation time to define the new configuration data is in the range of 63-153 ms, which can be prohibitive for some applications. The reconfiguration time for a new configuration data is not critical (around tens of μs) since only partial configuration is performed. As it can be seen on Fig. 1.d partial configuration enables to significantly save area compared to a generic implementation since in that case the architecture is specialized for each key. the security policy supported by the security module controllers are not explicitly presented in these papers. Figure 1 highlights that various solutions can be implemented for a same security primitive hence various area/throughput/reliability trade-offs can be considered. Agility enables to promote these trade-offs and then to adapt dynamically both performance and security to actual execution context. A last point which is important is related to power consumption. All previous studies did not deal with that point however for ambient system it is an essential feature. In [12], energy efficient solutions are proposed for the AES security primitive. In this case the important metric is Gbits/joule which is very relevant since ambient systems are mobile.

In conclusion of that part it is important for designers that have to build Configurable Security Module to be aware of all these trade-offs in order to promote agility and to meet with performance. Studies dealing thoroughly with configuration power consumption, secure communication links and security module controller policy are still required in order to propose secure Configurable Security Module and by extension secure systems. However agility provides many keys to build high-security/high-performance systems.

6 Conclusion

Configurable computing presents significant features to target high-security/high-performance ambient systems. However today these features are partially addressed and it is time to extend the vision of security using configurable computing since not only some parts but the whole system will be embedded within configurable systems. In this paper an analysis of major issues dealing with security at the hardware level using configurable computing is proposed. The goal of this paper is to stress that configurable computing is not just hardware accelerators for security primitives as most studies have focused on. Actually this point is part of the global system when dealing with configurable embedded computing. For that purpose two complementary views are proposed in order to guide the designer when facing with the difficult problem of system security and key aspects related to agility are presented and illustrated through the AES security primitive. Clearly there are still many issues to make security commonplace dealing with configurable computing and to define the overhead costs that imply security mechanisms at the hardware level but this paper aims to propose a first step toward a security design guide using configurable computing to meet with high-security/high-performance ambient system requirements.

References

1. Plessl, C., Enzler, R., Walder, H., Beutel, J., Platzner, M., Thiele, L., Troster, G.: The case for reconfigurable hardware in wearable computing. Personal and Ubiquitous Computing **7** (2003) 299–308

2. Xenakis, C., Merakos, L.: Security in third generation mobile networks. Computer Communications **27** (2004) 638–650
3. Guilley, S., Pacalet, R.: SoC security: a war against side-channels. Systeme sur puce electronique pour les telecommunications **59** (2004)
4. Cravotta, N.: Prying eyes. EDN (2002) http://www.edn.com/toc-archive/2002/20020926.html.
5. Standaert, F.X., tot Oldenzeel, L.V.O., Samyde, D., Quisquater, J.J.: Power analysis of FPGAs: How practical is the attack? In Heidelberg, S.V., ed.: international conference on Field-Programmable Logic and its Applications (FPL 2003). Volume LNCS 2778. (2003) 701–711
6. Anderson, R., Kuhn, M.: Tamper resistance - a cautionary note. In: Second USENIX Workshop on Electronic Commerce Proceedings, Oakland, California, USA (1996)
7. Wollinger, T., Paar, C.: Security aspects of FPGAs in cryptographic applications. In Rosenstiel, W., Lysaght, P., eds.: New Algorithms, Architectures, and Applications for Reconfigurable Computing. Kluwer (2004)
8. Elbirt, A., Yip, W., Chetwynd, B., Paar, C.: An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **9** (2001) 545–557
9. Dandalis, A., Prasanna, V.: An adaptive cryptography engine for internet protocol security architectures. ACM Transactions on Design Automation of Electronic Systems (TODAES) **9** (2004) 333–353
10. Gogniat, G., Wolf, T., Burleson, W.: Configurable security architecture for networked embedded systems. Technical report, ECE Department, University of Massachusetts, Amherst, USA (2004)
11. Bossuet, L., Gogniat, G., Burleson, W.: Dynamically configurable security for SRAM FPGA bitstreams. In: 11th Reconfigurable Architectures Workshop (RAW 2004), Santa Fé, New Mexico, USA (2004)
12. Schaumont, P., Verbauwhede, I.: Domain specific tools and methods for application in security processor design. (2002) 365–383
13. Gaj, K., Chodowiec, P.: Fast implementation and fair comparison of the final candidates for advanced encryption standard using field programmable gate arrays. In Springer-Verlag, ed.: RSA Security Conf. - Cryptographer's Trac, San Francisco, CA, USA (2001) 84–99
14. McLoone, M., McCanny, J.: High performance single-chip FPGA Rijndael algorithm implementations. In: Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001), Paris, France (2001) 65–76
15. Standaert, F.X., Rouvroy, G., Quisquater, J.J., Legat, J.D.: A methodology to implement block ciphers in reconfigurable hardware and its application to fast and compact aes rijndael. In: ACM/SIGDA 11th International Symposium on Field Programmable Gate Arrays (FPGA 2003), Monterey, California, USA (2003) 216–224
16. Saggese, G.P., Mazzeo, A., Mazzocca, N., Strollo, A.G.M.: An FPGA-based performance analysis of the unrolling, tiling, and pipelining of the AES algorithm. In Heidelberg, S.V., ed.: International Conference on Field-Programmable Logic and its Applications (FPL 2003). Volume LNCS 2778. (2003) 292–302
17. Hodjat, A., Verbauwhede, I.: A 21.54 Gbits/s fully pipelined AES processor on FPGA. In: IEEE Symposium on Field -Programmable Custom Computing Machines (FCCM 2004). (2004)
18. Standaert, F.X., Rouvroy, G., Quisquater, J.J., Legat, J.D.: Efficient implementation of Rijndael encryption in reconfigurable hardware: Improvements and design tradeoffs. In Springer, ed.: Cryptographic Hardware and Embedded Systems (CHES 2003). Volume Lecture Notes in Computer Science 2779., Cologne, Germany (2003) 334–350

19. Järvinen, K., Tommiska, M., Skyttä, J.: A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In: ACM/SIGDA 11th International Symposium on Field Programmable Gate Arrays (FPGA 2003), Monterey, California, USA (2003) 207–215
20. Karri, R., Wu, K., Mishra, P., Kim, Y.: Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **21** (2002)
21. McMillan, S., Cameron, C.: JBits implementation of the advanced encryption standard (Rijndael). In: International Conference on Field-Programmable Logic and its Applications (FPL 2001), Belfast, Ireland (2001)

FPL-3E: Towards Language Support for Reconfigurable Packet Processing

Mihai Lucian Cristea[1], Claudiu Zissulescu[1], Ed Deprettere[1], and Herbert Bos[2]

[1] Leiden University, The Netherlands
{cristea, claus,edd}@liacs.nl
[2] Vrije Universiteit Amsterdam, The Netherlands
herbertb@cs.vu.nl

Abstract. The FPL-3E packet filtering language incorporates explicit support for reconfigurable hardware into the language. FPL-3E supports not only generic header-based filtering, but also more demanding tasks such as payload scanning and packet replication. By automatically instantiating hardware units (based on a heuristic evaluation) to process the incoming traffic in real-time, the NIC-FLEX network monitoring architecture facilitates very high speed packet processing. Results show that NIC-FLEX can perform complex processing at gigabit speeds. The proposed framework can be used to execute such diverse tasks as load balancing, traffic monitoring, firewalling and intrusion detection directly at the critical high-bandwidth links (e.g., in enterprise gateways).

1 Introduction

There exists a widening gap between advances in network speeds and those in bus, memory and processor speeds. This makes it ever more difficult to process packets at line rate. At the same time, we see that demand for packet processing tasks such as network monitoring, intrusion detection and firewalling is growing. Commodity hardware is not able to process packet data at backbone speeds, a situation that is likely to get worse rather than better in the future. Therefore, more efficient and scalable packet processing solutions are needed.

It has been recognised that parallelism can be exploited to deal with processing at high speeds. A network processor (NP), for example, is a device specifically designed for packet processing at high speeds by sharing the workload between a number of independent RISC processors. However, for very demanding applications (e.g., payload scanning for worm signatures) more power is needed than any one processor can offer. For reasons of cost-efficiency it is infeasible to develop NPs that can cope with backbone link rates for such applications. An attractive alternative is to use a reconfigurable platform such as an FPGA that exploits parallelism at a coarser granularity.

We have previously introduced the efficient monitoring framework Fairly Fast Packet Filters (FFPF) [1], that can reach high speeds by pushing as much of the work as possible to the lowest levels of the processing stack (see Fig. 1.b). The NIC-FIX architecture [2] showed how this monitoring framework could be extended all the way down to the network card. To support such an extensible programmable environment, we introduced the special purpose language known as FPL-3.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 82–92, 2005.

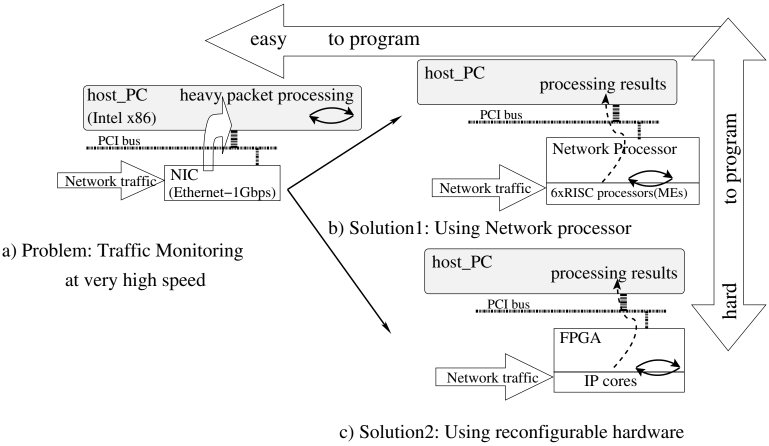


Fig. 1. Moving to special purpose embedded systems

In this paper, we exploit packet processing parallelism at the level of individual processing units (FPGA cores) to build a monitoring architecture: NIC-FLEX(see Fig. 1.c). Incoming traffic is stored in fast off-chip memory, wherefrom it is processed by multiple FPGA cores in parallel. The processing results are first stored in a very fast local memory and then passed, on demand, to a higher level (e.g., user space tools). The main contribution of this paper consists of a novel language that explicitly facilitates parallelisation of complex packet processing tasks: FPL-3E. Also, with NIC-FLEX we extend the FFPF architecture upwards with specific packet processing support to create a flexible and fast filtering platform. Experiments show NIC-FLEX to be able to handle complex tasks at gigabit line-rate.

This paper builds on the idea of extensible system-on-programable-chip that was advocated by Lockwood *et al.* in [3] for firewalling. However, we use it to provide a generic high-speed packet processing environment by using the Compaan/Laura tool chain [4, 5] that automatically transforms a user code into synthesizable VHDL code that targets a specific FPGA platform.

The remainder of this paper is organised as follows. In Section 2, the architecture of the packet processing system and its supporting language are presented. Section 3 is devoted to the implementation details. The proposed architecture is evaluated in Section 4. Related work is discussed throughout the text and summarised in Section 5. Finally, conclusions are drawn and options for future research are presented in Section 6.

2 Architecture

2.1 High-Level Overview

At present, high speed network packet processing solutions need to be based on special purpose hardware such as dedicated ASIC boards or network processors (see Fig. 1.b). Although faster than commodity hardware (see Fig. 1.a), solutions based even on these platforms are surpassed by the advances in reconfigurable hardware systems, e.g., FPGAs.

To counter this packet processing trend we propose the solution shown in Fig. 1c, which consists of mapping the user's program onto hardware, processing the incoming traffic efficiently, and then passing the processing results back to the user.

The software architecture is comprised of three main components (see Fig. 2). The first component ① is a high level interface to the user and the kernel space of an Operating System (e.g., Linux) and is based on the Fairly Fast Packet Filter (FFPF) [1] framework. The second component ② is the FPL-compiler interface between the first and the last components. The compiler takes a program written in a packet processing language ⓐ and generates a code object ⓑ for the lowest level of processing: Reconfigurable hardware. The third component ③ is a synthesiser tool that maps specific processing algorithms onto an FPGA platform and is based on the Laura tool.

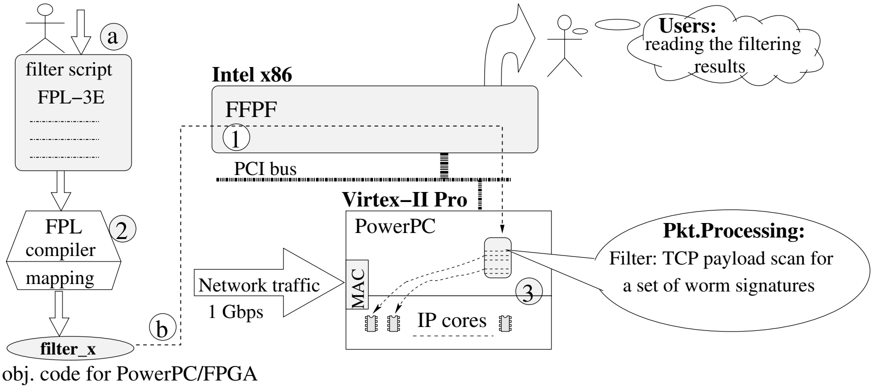


Fig. 2. Packet processing architecture

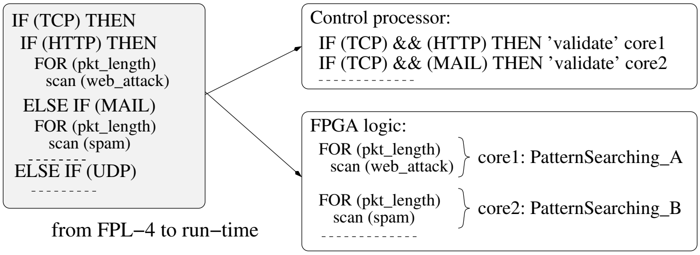
2.2 The FFPF Software Framework

FFPF was designed to meet the following challenges: (1) monitor high-speed links and scale with future link rates, (2) offer more flexibility than existing packet filters, and (3) provide a migration path by being backward compatible with existing approaches (notably `pcap`-based applications [6]). The FFPF framework supports userspace programs, kernel, the IXP1200 network processor, or a combination of the above. FFPF now extends also to reconfigurable hardware by introducing explicit support for FPGA in the FFPF programming language (FPL-3E).

2.3 The FPL-3E Language and the FPL-Compiler

As our architectural design relies on explicit hardware support, we needed to introduce this functionality into our framework. With FPL-3E, we adopted a language-based approach, following our earlier experiences in this field. We designed FPL-3E specifically with these observations in mind: First, there is a need for executing tasks (e.g., payload scanning) that existing packet languages like BPF [6] or Snort [7] cannot perform. Second, special purpose devices such as network processors or FPGAs can be quite complex and thus are not easy to program directly. Third, we should facilitate on-demand extensions, for instance through hardware assisted functions. Finally, security issues such as user authorisation and resource constraints should be handled effectively. The previous version of the FPL-3E language, FPL-3 [8], addressed many of these concerns. However, it lacked features fundamental to reconfigurable hardware processing like resource partition and parallel processing.

We will introduce the language design with an example. First, a simple program requiring a high amount of processing power is introduced in Figure 3. Then, the same example is discussed through multiple 'mapping' cases by using the FPL-3E language extensions in Figures 4, 5.



```
IF (TCP) THEN
  IF (HTTP) THEN
    FOR (pkt_length)
      scan (web_attack)
  ELSE IF (MAIL)
    FOR (pkt_length)
      scan (spam)
     --------
 ELSE IF (UDP)
  ---------
```

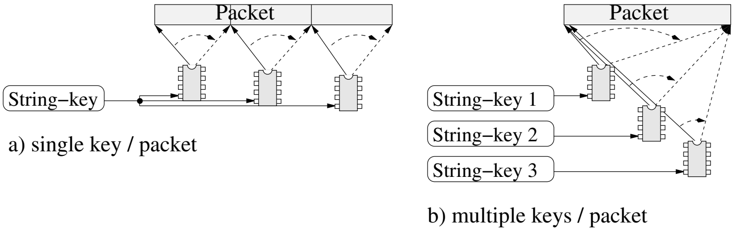
from FPL–4 to run–time
Control processor:
IF (TCP) && (HTTP) THEN 'validate' core1
IF (TCP) && (MAIL) THEN 'validate' core2
FPGA logic:
FOR (pkt_length) scan (web_attack)
core1: PatternSearching_A
FOR (pkt_length) scan (spam)
core2: PatternSearching_B

Fig. 3. Packet processing example

As Figure 3 shows, the FPL-3E compiler translates the program into multiple output objects, one for a control processor (ASIC embedded into the FPGA) and a second one for the FPGA reconfigurable hardware (logic that contains multiple cores). The FPGA cores consist of specific heavy computation algorithm implementations (e.g., pattern searching) that are interconnected in such way as to achieving an optimal processing path as we show later in this section. Besides the parallelism built into the logic, we note that the task from embedded control processor runs itself in parallel with the FPGA logic. The control code is mostly composed of nested IF statements used for result validation and, therefore, the processing speed of the control processor is high enough to keep up with the high speed FPGA data processing.

Note that the requirement to perform complex packet processing at Gbps line rate means that each packet has to be processed within a very limited time budget – a basic task. When a task requires a large amount of *per-packet* processing power (e.g., a full packet scan for a worm), it becomes infeasible to perform this task on a single processing unit when network speeds go up. Thus, we give the same example mapped using various techniques for parallel processing environment. For the sake of simplicity we limit our granularity to three levels.

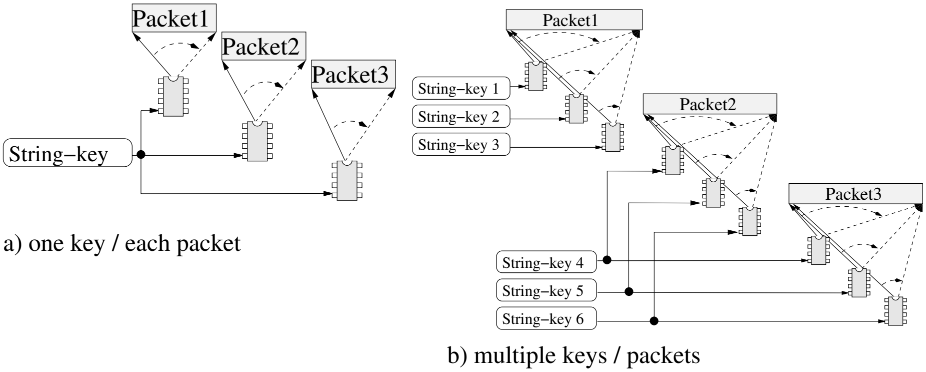
A basic processing task consists of searching through the whole packet payload data for a string (e.g., a worm signature) and it is performed by a processing unit implemented in hardware. When the task overloads the processing unit, then this task can be distributed across three hardware units in parallel, using one search key per packet, as shown in Figure 4.a, or multiple keys per packet (see Fig. 4.b), or a combination of both techniques. In the first configuration, the required number of cycles is reduced with the number of hardware devices instantiated – three in our example, as the same string is searched on different parts of the packet. The second approach allows us to search in parallel three signatures on the same packet at a cycles cost of one. However, when the receiving rate is higher than the processing abilities given by 'one packet' approach, we can process multiple packets in parallel (depth-processing), as illustrated in Figure 5.



a) single key / packet

b) multiple keys / packet

Fig. 4. Packet processing techniques



a) one key / each packet

b) multiple keys / packets

Fig. 5. Multi-packet processing techniques

The FPGA technology gives us enough flexibility to choose for one or a mix of the above mentioned approaches. It also provides a long-term platform life by its ease-to-extend with new algorithm implementations, such as IP cores specifically designed for pattern matching, regular expressions, protocol recognition, etc. This support will address future issues like adaptivity to new protocols (e.g., peer-to-peer). The limitation is given only by the hardware capacity (that nowadays goes beyond our needs) and the compiler abilities to perform such complex mapping from a simple and 'natural' programming language: FPL-3E.

2.4 The Compaan/Laura Tool Chain

The FPGA platform is a highly parallel structure suitable to accommodate algorithms that exploit this parallelism. Although this texture is the key to take advantage of the platform, the commonly used imperative specification programming languages like C, Java or Matlab are hard to compile to parallel FPGA specifications. In general, specifying an application in a parallel manner is a difficult task. Therefore, we used the *Compaan* Compiler [4] that fully automates the transformation of sequential specification to an input/output equivalent parallel specification expressed in terms of a so-called Kahn Process Network (KPN). Subsequently, the *Laura* tool [5] takes as its input this KPN specification and generates synthesizable VHDL code that targets a specific FPGA platform. The Compaan and Laura tools together realise a fully automated design flow that maps sequential algorithms onto a reconfigurable platform. We use this design tool chain to implement our computational intensive cores such as pattern matching algorithms. Thus, in FPL-3E, we separate the control intensive tasks from data intensive tasks. These last tasks are automatically analysed and mapped onto an FPGA platform thereby exploiting the inherit parallelism of the data processing algorithms.

3 Implementation Details

3.1 The FPL-3E Language

The FFPF programming language (FPL) was devised to give the FFPF platform a more expressive packet processing language than previously available. The FPL-3E syntax is summarised in Figure 6. It supports all common integer types and allows expressions to access any field in the packet header or payload in a friendly manner.

The latest version (FPL-3) conceptually uses a register-based virtual machine, but compiles to fully optimised object code. FPL-3 supports commodity PCs and NPs (further implementation details available in [8]). We now introduce its direct descendant, FPL-3E, which extends FPL-3 with constructs for reconfigurable hardware processing.

EXTERN() Construct. This was introduced in FPL-3 to support the ‘extensibility’ system feature. We extend it with support for reconfigurable hardware devices (FPGAs). `EXTERN(name,input,output,hw_depth)` tells the compiler that the task needs the help of the specified core ‘name’ to process the current packet according to ‘input’ parameters and place the processing results in the ‘output’. ‘hw_depth’ is an optional parameter for advance users that want to ‘force’ the compiler to use a certain amount of hardware units for parallel packet processing. By default, the compiler estimates this parameter accordingly to the incoming traffic rate and the available hardware resources (given at compile time).

3.2 The FPL-3E Compiler

The FPL-3E is a source-to-source compiler. Like its predecessors, it generates straight C target code that can be further handled by any C compiler. Programs can therefore benefit from the advanced optimisers in the Intel μEngine C compiler for IXP devices, `gcc` for commodity PCs and Xilinx ISE for Xilinx’s FPGAs. As a result, the object code will be heavily optimised even though we did not write an optimiser ourselves.

Moreover, the FPL-3E compiler uses a heuristic evaluation of the hardware instances needed to reach the system goal (e.g., the line rate is 1Gbps). The evaluation is based on the workload given by one hardware instance to perform the user's program and a critical point where the performances fall down because of some heavy computation like signature length, or packet size. For example, in Figure 5.b, assuming that the user's program performs checking of six signatures, but three of them are known as much longer than the others, the compiler duplicates the hardware units, accordingly, in order to achieve a well balanced workload of the whole system.

operator-type	**operator**
Arithmetic	+, -, /, *, %, --, ++
Assignment	=,*=, /=, %=, +=, -= <<=, >>=, &=, ^=, |=
Logical / Relational	==, !=, >, <, >=, <=, &&, ||, !
Bitwise	&, |, ^, <<, >>

statement-type	**operator**

if/then/else	IF (expr) THEN stmt1 FI
	ELSE stmt2 FI

for()	FOR (initialise; test; update) stmts; BREAK; stmts; ROF
return a value	RETURN (val)
external function	INT EXTERN(name,input, output) or INT EXTERN(name,input, output,hw_depth)

Data type	**syntax**
Register n	R[n]
Memory location n	M[n]
Packets access:	
-byte $f(n)$	PKT.B[$f(n)$]
-word $f(n)$	PKT.W[$f(n)$]
-double word $f(n)$	PKT.DW[$f(n)$]
-bit m in byte n	PKT.B[n].U1[m]
-nibble m in byte n	PKT.B[n].U4[m]
-bit m in word n	PKT.W[n].U1[m]
-byte m in word n	PKT.W[n].U8[m]
-bit m in dword n	PKT.DW[n].U1[m]
-byte m in dword n	PKT.DW[n].U8[m]
-word m in dword n	PKT.DW[n].U16[m]
-macro	PKT.macro_name
-ip proto	PKT.IP_PROTO
-ip length	PKT.IP_LEN
-etc.	customised macros

Fig. 6. FPL-3E language constructs

3.3 Control Processor and FPGA Cores

In modern FPGAs there are embedded from one to four hard cores control processors (e.g., PowerPC or ARM), that are suitable to map the control part of our algorithms. The data intensive tasks are mapped directly in hardware (IP cores) using the Compaan/Laura tool chain. The IP cores communicate with the control processor using a set of registers to set some run-time parameters (e.g., the packet length or the searched key-strings).

To study the feasibility of using the Compaan/Laura tool chain in the Networking world we compiled in hardware a search algorithm. The Matlab program for this algorithm is shown in Figure 7. The bytes of the packet (e.g., *pkt*()) are compared with the content of a signature string (e.g., *sig*()). If the signature is present in the packet, then the value of the *c* variable is equal to the length of the searched string.

The program illustrated in Figure 7 has been rewritten to match the requirements of the Compaan/Laura tool chain. Additionally, we instructed our tool to generate a design that compares eight characters in parallel. The hardware network of processors

```
for i = 7: 1: PackSize,
  c = 0;
  for j = 1: 1: StringLength,
    if sig(j) = pkt(i),
       c = c + 1;
    end
  end
  if c = StringLength,
    print "Found!"
  end
end
```

Fig. 7. Simple Search Algorithm

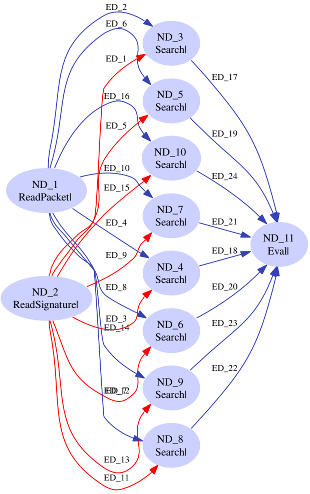


Fig. 8. Processor Network of the simple algorithm

Table 1. Experimental results

Packet Length	String Length	Clocks/Workload	Slices	Frequency (MHz)
64	8	77	2035	101

is depicted in Figure 8. Each bubble represents a hardware processor and each arch a communication channel between two processors. The *ReadPacket* processor feeds our network with packet bytes from a MAC network interface. The *Search* processor implements the character-wise searching, the result of 'a search' is evaluated by the *Eval* processor, which is also our write interface toward external devices.

Table 1 gives the hardware results of the FPGA implementation of the algorithm given in Figure 7. The experiment has been done using Synplify and ISE Xilinx 6.2 for the Virtex II-6000 platform. The hardware is capable of doing an eight character string search in a variable packet size. The required number of cycles for a variable packet size and eight characters search string is $cycles = 13 + PackSize$.

In our example, the length of the search string is fixed to eight characters. However, the string size can be changed at compile time while its content may be changed at run-time.

4 Evaluation

Given the pattern matching algorithm result (see Table 1) for one search-key per packet, we extrapolate to other case studies as already illustrated in Figure 5. In Figure 9 is shown how the performance of one key per packet approach (1key/1hw) scales up by increasing the use of hardware units (1key/3hw) in parallel.

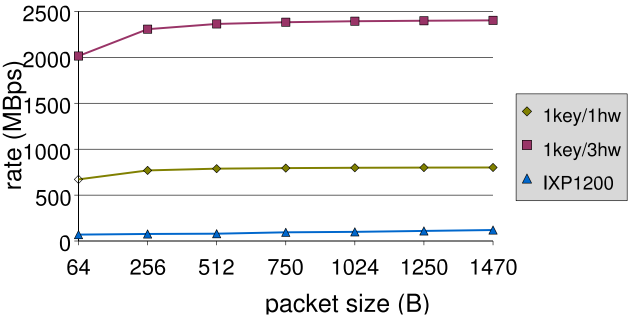


Fig. 9. FPGA vs. NP processing results

The processing result of a full packet payload pattern search filter performed by a 1Gbps generation network processor (Intel IXP1200) is also shown in Figure 9. Therefore, making a comparison between an FPGA implementation and a network processor implementation, it can be seen that a complex filter (such as a pattern searching algorithm) performed by a NP is surpassed by even a single IP core implementation.

Note that the relatively small amount of hardware resources used for this implementation (ca. 6% for a Virtex II-6000) allows us to map more than one search engine into a FPGA platform.

5 Related Work

The usage of accelerated cores has been done in the Molen project [9] by annotating a PowerPC processor with a set of multimedia instructions that are accelerated in hardware. Our approach focuses on the networking applications algorithms. Thus, our hardware accelerated cores perform coarse grain computations (e.g., pattern matching). Additionally, the number of processing elements for a particular task can be set-up at compile time based either on user demands or on the built-in compiler heuristic estimator about the workload requirements.

Using reconfigurable hardware for increased packet processing efficiency was previously explored in [10] and [11]. Our architecture differs in that it provides explicit language support for this purpose. As shown in [12], it is efficient to use a source-to-source compiler from a generic language (Snort Intrusion Detection System) to a back-end language supported by the targeted hardware compiler (e.g., Intel μEngineC, PowerPC C, VHDL). We propose a more flexible and easy to use language as front-end for users. Moreover, our FPL-3E language is designed and implemented for heterogeneous targets in a multi-level system.

The SCAMPI architecture also pushes processing to the NIC [13]. It assumes that hardware can write packets immediately into host memory (e.g., by using DAG cards [14]) and implements access to packet buffers through a userspace daemon. SCAMPI does not support user-provided external functions, powerful languages such as FPL-3E.

6 Conclusions and Future Work

This paper presented the NIC-FLEX packet processing environment and its FPL-3E programming language, which enable users to process network traffic at high speeds by mapping of their programs onto reconfigurable hardware (FPGA). A program is mapped by loading IP cores generated using the Compaan/Laura approach to implement the data intensive tasks in hardware. Currently, this task is performed by an engineer and thus, the user cannot generate its own tasks in hardware. However, we supply the FPL framework with a wide range of hardware cores to overcome the need for different cores. All these cores are annotated with performace numbers such that the FPL-3E environment computes the right work balance, based on a heuristic evaluation. This heuristics may be ignored by the user and replaced with its own evaluation. The experimental results show that NIC-FLEX can outperform traditional packet filters by processing at Gbps line rate.

In the future, we plan to extend NIC-FLEX with a management environment that can take care of object code loading and program instantiation.

Acknowledgements

This work was supported by the EU SCAMPI project IST-2001-32404, while Intel donated the network cards and Xilinx donated the development kit.

References

1. Bos, H., de Bruijn, W., Cristea, M., Nguyen, T., Portokalidis, G.: FFPF: Fairly Fast Packet Filters. In: Proceedings of OSDI'04, San Francisco, CA (2004)
2. Nguyen, T., de Bruijn, W., Cristea, M., Bos, H.: Scalable network monitors for high-speed links: a bottom-up approach. In: Proceedings of IPOM'04, Beijing, China (2004)
3. Lockwood, J.W., Neely, C., Zuver, C., Moscola, J., Dharmapurikar, S., Lim, D.: An extensible, system-on-programmable-chip, content-aware Internet firewall. In: Field Programmable Logic and Applications (FPL), Lisbon, Portugal (2003) 14B
4. Kienhuis, B., Rypkema, E., Deprettere, E.: Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In: Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES), San Diego, USA (2000)
5. Zissulescu, C., Stefanov, T., Kienhuis, B., Deprettere, E.: LAURA: Leiden Architecture Research and Exploration Tool. In: Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL'03), Lisbon, Portugal (2003)
6. McCanne, S., Jacobson, V.: The BSD Packet Filter: A new architecture for user-level packet capture. In: Proceedings of the 1993 Winter USENIX conference, San Diego, Ca. (1993)
7. Roesch, M.: Snort: Lightweight intrusion detection for networks. In: Proceedings of the 1999 USENIX LISA Systems Adminstration Conference. (1999)
8. Cristea, M.L., de Bruijn, W., Bos, H.: Fpl-3: towards language support for distributed packet processing. In: Proceedings of IFIP Networking 2005 (accepted for publication), Waterloo, Canada (2005)
9. Vassiliadis, S., Wong, S., Gaydadjiev, G.N., Bertels, K., Kuzmanov, G., Panainte, E.M.: The molen polymorphic processor. IEEE Transactions on Computers (2004)

10. Anto, D., Koenek, J., Minakov, K., ehk, V.: Packet header matching in combo6 ipv6 router. Technical Report 1, CESNET (2003)
11. Clark, C., Lee, W., Schimmel, D., Contis, D., Kone, M., Thomas, A.: A hardware platform for network intrusion detection and prevention. In: The 3rd Workshop on Network Processors and Applications (NP3), Madrid, Spain (2004)
12. Charitakis, I., Pnevmatikatos, D., Markatos, E.: Code generation for packet header intrusion analysis on the ixp1200 network processor. In: SCOPES 7th International Workshop. (2003)
13. Polychronakis, M., Markatos, E., Anagnostakis, K., Oslebo, A.: Design of an application programming interface for ip network monitoring. In: IEEE/IFIP NOMS, Seoul (2004)
14. Cleary, J., Donnelly, S., Graham, I., McGregor, A., Pearson, M.: Design principles for accurate passive measurement. In: Proceedings of PAM, Hamilton, New Zealand (2000)

Flux Caches: What Are They and Are They Useful?

Georgi N. Gaydadjiev and Stamatis Vassiliadis

Computer Engineering, EEMCS, TU Delft, The Netherlands
{G.N.Gaydadjiev, S.Vassiliadis}@ewi.tudelft.nl
http://ce.et.tudelft.nl

Abstract. In this paper, we introduce the concept of flux caches envisioned to improve processor performance by dynamically changing the cache organization and implementation. Contrary to the traditional approaches, processors designed with flux caches instead of assuming a hardwired cache organization change their cache "design" on program demand. Consequently program (data and instruction) dynamic behavior determines the cache hardware design. Experimental results to confirm the flux caches potential are also presented.

1 Introduction

To improve processor performance numerous cache organizations have been proposed (and some of them implemented) in the past. All well known cache organizations can be divided in two classes: A) static approaches, e.g. victim [1, 2] [1], column associative [4], skewed-associative [5] and assist [6] caches; and B) adaptive designs, e.g split temporal/spatial [7], dual data [8], reconfigurable [9] and configurable line size [10] caches. The first group relies on time invariant design improvements, while the second one aims on trivial cache organization changes according to some running application requirements. We envision a third approach, termed *flux caches*, based on demand driven cache designs and implemented using for example reconfigurable technologies.
Reconfigurable hardware extensions of general purpose processors (GPP) have been mainly focusing on accelerating frequently used code in hardware [11, 12, 13]. Such hardware/software repartitioning usually leads to drastic changes in cache behavior since the application temporal and spacial locality is mainly accounted on highly iterative loops that form the primary subjects for hardware implementation. While dealing with the aforementioned effects did not get unnoticed [14], using on-demand hardware designs to improve the GPP memory sub-system seems to lack attention from the research community. In this paper depending on expected execution benefits of a single program (or a subsection of a program), memory sub-system designs are changed on demand. If during program execution (or before the execution of a program) it is found or expected that a different cache organization is beneficial then a new cache design is (dynamically) installed in hardware. In essence our approach allows on demand L1, L2 cache designs where all cache parameters (e.g. associativity, total size, line size,

[1] This is the earliest work on victim caches presented before the widely recognized victim cache paper of Jouppi [3].

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 93–102, 2005.

replacement policy, victim cache addition etc.) can be adjusted. Using reconfigurable technologies we show how to incorporate our approach with no need for architectural changes. We target an existing processor platform [15] and show that dynamic cache design can be transparently done with no architectural (ISA) changes.

The remainder of this paper is organized as follows. Section 2 introduces the flux caches and how they map to the MOLEN machine organization. Section 3 reviews the most relevant related work. In Section 4 the simulation framework for this study and the performance results are described. Finally, the discussion is concluded in Section 5.

2 Flux Caches Organization and Implementation

It is envisioned that different programs have unique cache requirements that can be satisfied by alternative cache organizations. Support for such flexibility is expected to exploit significant improvements in application execution times. For example, let us consider two applications of different kind running on the same embedded (e.g. in a mobile phone) processor. The first one is a digital video processing algorithm with predominantly streaming (spacial locality) memory accesses. Let the second application be a Java Virtual Machine (JVM) with heavy temporal locality memory accesses. Obviously the system designer is confronted with a dilemma considering the fact that drowsy behavior for both cases is considered unacceptable. Coming up with a cache design that works optimally for both applications is rather difficult. Let assume instead that both applications can at advance (before they start) set up a cache design that will best fit their particular memory requirements. This is not such a non-realistic scenario since in the majority of the cases the user will never watch a football match and play a strategy game at the same time. In such a system, different cache designs coexist in time with their corresponding applications and can be optimized according to the specific demands.

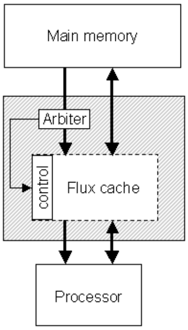


Fig. 1. Flux cache

Flux caches are fully customizable memories, possibly implemented in reconfigurable hardware, that can be installed on demand before or during program execution. Hardware implementations of arbitrary cache design can be instantiated under software or hardware control at runtime and are pre-determined ”off-line” at hardware/software co-design stage using application partitioning, monitoring, profiling etc. Generally speaking the flux cache mechanism would require additional ISA support [2] to enforce the intended cache design and will introduce some reconfiguration overhead. The flux cache organization is depicted in Figure 1.

The *Arbiter* will partially decode the instructions received from the instruction fetch unit and issue the flux cache instructions to the *control* unit. The control unit is responsible for loading the cache configuration code from memory and instruction / data paths consistency. The envisioned operations support consists only of a single *put*

[2] But not always as it will be shown later by using the MOLEN polymorphic processor [15].

phase. During this phase, the flux cache is configured to the intended hardware organization. More precisely, a bitstream is loaded from the main memory into the local configuration memory. This concept requires one-time architectural extension by a single instruction. The **put** instruction that initiates the flux cache configuration has the following format: *put <address>*. The *address* is a memory location the first element of the configuration bitstream is to be loaded from. Parameters of the cache are usually implicit as in the example presented hereafter, explicit calls can also be envisioned. The *put* phase is initiated by the arbiter after detection of a **put** instruction and has to be interrupted right after the hardware configuration is completed. This can be achieved only by proper configuration bitstream termination. There exist two different approaches (using special operation at the end or by defining the configuration code length at the beginning) both with their advantages and shortcomings.

Assuming the case (different from the aforementioned two applications example) of a single application with clearly defined regions with predominant spacial or temporally localities executing on a machine augmented with a flux cache (Figure 2). The original GPP execution code sequence is augmented with **put** instructions at the positions different cache organization is needed. The decision on cache type, size and configuration is left to the system designer since he/she is expected to understand the targeted applications behavior. The cache selection process can be supported by profiling, cache simulation and/or dynamic program monitoring. In addition, the latter process can be fully automated and integrated in the automated design tools. The **put** instruction will be redirected to the control unit and interpreted. More precisely, the configuration microcode located at the targeted address will be loaded into the configuration memory to ensure the flux cache hardware structure. After the cache reconfiguration is completed (and all *valid* tags of the "new" cache are invalidated) the execution of the GPP will continue from the next instruction following the **put**. In order to reduce the penalty of such execution stalls various prefetch and partial configuration techniques [16] and concurrent loading can be applied. Please note that after complete reconfiguration, the "new" cache will be "empty" and the cold-start effects have to be taken into consideration (keeping "old" filled caches, prefetching designs to fill caches and partial flux cache designs may help). The flux caches can be realized using existing technology, i.e. Virtex II Pro platform FPGA from Xilinx. The only constraint on the targeted technology is partial reconfiguration support.

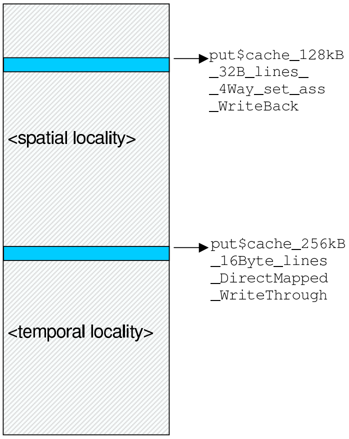


Fig. 2. Single program execution

To show the flux cache feasibility we assume reconfigurable implementation and the MOLEN paradigm. The MOLEN machine organization consists of two main components: the Core Processor (CP), usually a general purpose processor, and the Reconfigurable Processor (RP).

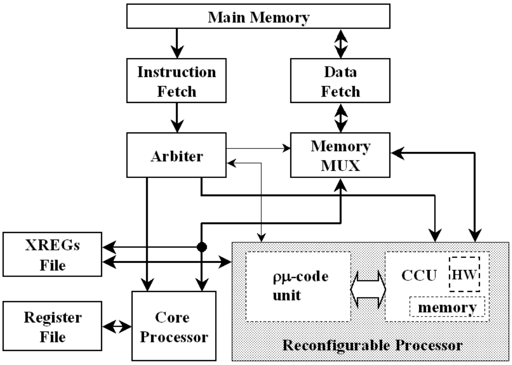


Fig. 3. MOLEN organization

The application's division in a hardware and a software part is directly mappable to the above two units. The execution flow redirection is performed by the Arbiter using partial instruction decoding. In respect to the Core Processor original ISA, MOLEN requires only an one-time extension with four and up to eight instructions dependent on the specific implementation [16]. To perform the actual reconfiguration of the CCU, reconfiguration microcode is loaded into the ρμ-code unit. This stage is also referred to as the **set** phase. The **execute** phase is responsible for the actual operation execution on the CCU, and is performed by running the *execution microcode*. It is important to emphasize that both the **set** and **execute** phases do not specify any pre-defined hardware operation to be performed. Instead, the **pset**, **cset** and **execute** instructions (*reconfigurable instructions*) directly point to the memory location where the reconfiguration or execution microcode is located. The hardware/software communication is supported by the Exchange Registers bank and performed through the **movtx** and **movfx** MOLEN instructions. As depicted on Figure 4, flux caches can be implemented under a simplified MOLEN scenario (only flux caches no CCUs). Cache coherence logic may be needed if for example the core processor employs L1 caches. The *put* flux cache phase is functionally equivalent to the *set* phase in MOLEN. All MOLEN configuration microcode termination and prefetching techniques [13] are directly applicable to flux caches. Said this we can use the MOLEN **set** instruction for **put** emulation. The execution phase with its supporting MOLEN instructions and functional modules is no longer needed for the flux cache implementation case. This allows the overal system organization to be reduced significantly. First of

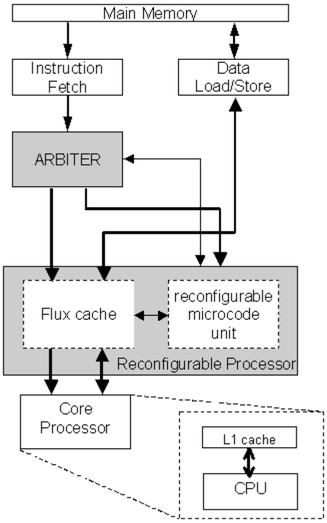


Fig. 4. A flux cache implementation

all the data memory Multiplexer / Demultiplexer can be avoided due to the absence of CCU that will perform data accesses. The exchange registers bank and the two *move* instructions for data exchange between the core and the reconfigurable processors are not required. The sequential consistency model, inherent to MOLEN, will naturally arbitrate the execution of the core processor code with the flux cache reconfiguration times. This leads to a very minimal but still completely functional flux cache implementation that will decrease the complexity (and the overall overhead) of the MOLEN functional blocks. In reality, the arbiter and the simplified ρμ-code unit can be combined into a single module that handles the flux cache reconfigurations. It is to be noted that code running on such simplified MOLEN instantiation will be binary compatible with any other MOLEN implementation. In the opposite direction, however, additional fix up code and exception handling may be needed to cope with all not implemented MOLEN (e.g. *mov* and *execute*) instructions.

3 Related Work

The flux caches allow their internal structure to be ”redesigned” at any given moment during the execution time. This is the reason why only the time variant cache proposals (as introduced in Section 1) will be considered hereafter.

The ”reconfigurable caches” introduced by Ranganthan et. al. [9] divide the available cache memory into several partitions that may be used to support applications usually unable to exploit conventional caches in an optimal way. As example the multimedia applications with their streaming nature are used. Although named reconfigurable, this proposal is just an extension of the conventional set-associative and direct mapped cache designs to support a limited number of partitions that are dynamically selectable. In addition, special ISA support may be required to control repartitioning (in case the software controlled approach is used). Our proposal differs in two aspects: first we do not impose any limitation on the number of possible cache configurations; and second very limited or no additional ISA support (as in the case of the MOLEN processor) is required to indicate the intended configuration.

The Split Temporal/Spatial (STS) caches [7] employ two cache sub-systems: one for ”temporal” data and another for ”spacial” data. The main idea is that handling data with temporal locality in a ”spacial” way, e.g. prefetching its neighboring addresses is usually counterproductive. This leads to data classification into two sub-groups, each to be handled separately by the corresponding cache. Such classification can be performed on compile / profile or run-time. Two ways to express this to the hardware are envisioned: by ISA extension or by tagging. The flux caches differ from STS caches in the following way. First, we allow instruction and data cache modifications compared to data cache only target of the STS caches. Second, we do not require and additional ISA modifications or tag bits to implement similar functionality. STS caches can be implemented in flux caches in a straight forward way by using the MOLEN *pset* or *execute* instructions to distinguish between ”temporal” and ”spacial” data.

The Dual Data Cache (DDC) bears some similarities with STS. Like STS, it has two separate modules to deal with data of different locality. The data allocation, however, is more sophisticated and one additional *bypass* mode is introduced. The memory instructions are tagged as in STS with the difference that five different data types are distinguished. As in the case of STS caches our proposal differs in its flexibility concerning the instruction cache and its zero overhead ISA support.

The configurable line size caches proposed by the University of California, Riverside [10] focus mainly on the cache memory energy consumption. This work covers static selection of the cache line size early in the embedded system design process. The assumption is that an embedded system will execute only a pre-defined (and hence very limited) set of applications during its operational lifetime. Later ongoing research of the same group [14] reported dynamic configuration during run-time. However, again only a very limited number of cache configurations is supported. Our proposal does not impose such restriction on the system designers, allowing them to introduce changes later (even in the field) when new applications have been added or existing one should be upgraded (e.g. using MPEG4 instead of MPEG2). The above list of related work is not complete but to our knowledge representative. The reason of not including all previous approaches is the significant number of publications on the topic and space limitations. Proposals such as software managed data caches (implemented in HP PA-7200 CPU) [17] or Veidenbaum's et. al. dynamic cache line size adaptation [18] are not considered in details due to some similarities with DDC and the work from UC Riverside respectively.

All of the proposals reported in the publicly available literature do focus on organizing the available cache memory in a number of pre defined ways, mainly in respect to associativity and cache line size. In our proposal the only restriction known is the available reconfigurable hardware resources (e.g. on-chip SRAM size) that may limit the overal cache size. All remaining cache parameters, e.g. replacement strategy, prefetching and write back policy, can be adjusted to the targeted application in order to gain optimal performance. In addition, our proposal does not limit the system designer to the conventional cache architectures and provides him with means to utilize (and/or evaluate) unique approaches, e.g. stream caches, or even design and apply completely customized memory sub-system (e.g. 2-D rectangular memory [19]).

4 Simulation Framework, Methodology and Results

We studied the potential benefits of reconfigurable caches using dinero IV [20], a trace driven cache simulator that models the first two levels of the memory hierarchy. We share the opinion that statements about cache performance can be based only on trace-driven simulation or direct measurements [21]. The former method is slow and has significant demands on storage capacity, while the latter is fast but prohibitively expensive. Since our study is about relative cache performance, a non-functional simulator such as dinero is considered sufficient. The application traces for this study where obtained using the SimpleScalar 4.0 simulator [22] modified to generate dinero style memory traces. The traces where generated in an in-order execution fashion. Only the three basic memory access types where implemented: data read, data write and instruction fetch. This fact, however, does not have any influence on the generality of the reported results. The targeted applications of interest where multimedia.

Table 1. Benchmarks used in this study

benchmark	Description	Input
gsmenc	GSM speech encoding (toast)	clinton.pcm
gsmdec	GSM speech decoding (untoast)	clinton.pcm.gsm
adpcmenc	ADPCM speech encoding (rawcaudio)	clinton.pcm
adpcmdec	ADPCM speech decoding (rawdaudio)	clinton.adpcm
mpegenc	MPEG-2 video encoding (four 352x240 frames IBBP)	mei16v2.yuv
mpegdec	MPEG-2 video decoding (video stream to YUV)	mei16v2.m2v
cjpeg	JPEG encoding (1024x630 3-band image)	rose16.ppm
djpeg	JPEG decoding (1024x630 3-band image)	rose16.jpg
epicenc	EPIC encoding (unepic) (512x512 grayscale image)	test.image.pgm.E
epicdec	EPIC decoding (epic) (512x512 grayscale image)	test_image.pgm
g721enc	G721 speech compression	clinton.pcm
g721dec	G721 speech decompression	clinton.g721

This is the reason for selecting a representative set of benchmarks and corresponding data sets from the UCLA MediaBench [23] suite as summarized in Table 1. We targeted set of benchmarks that cover audio, video, images and speech data processing that is assumed to represent the application domain for our study. We have simulated many different L1 caches to explore the impact of various cache parameters on the miss ratio. All the simulation and data collection work was automated using a script that did attempt local and global minimum determination in the reported miss ratios. Sophisticated algorithms for optimal cache selection are outside the scope of the current study.

We do realize that some of the synthetic benchmarks used may not truly represent a real-life multimedia application. For example, the gsm pair (also known as toast/untoast) consists mainly of highly iterative functions that rely on the *register* keyword for speed up optimizations. In our case (SimpleScalar architecture and gcc compiler) unrealistically low data miss ratios are expected for those benchmarks. On the other hand, such situation forms a worst case scenario for evaluation of the proposed cache organization.

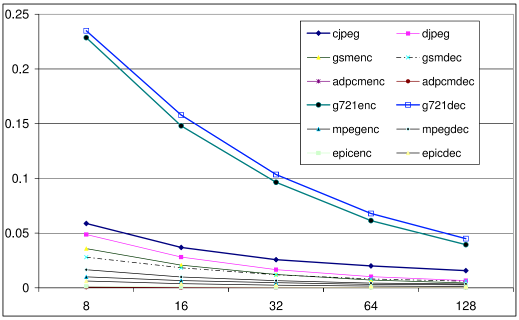
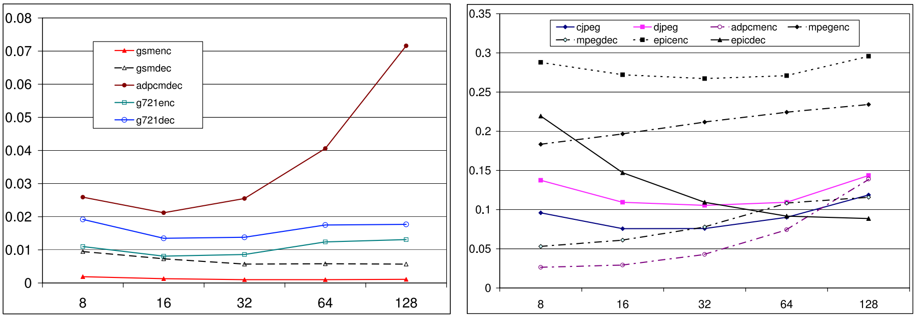


Fig. 5. I-cache miss ratios vs line size

In an attempt to evaluate the optimal configuration for the targeted benchmark set under a flux cache scenario, a variety of cache configurations where simulated. They all differ in overall cache sizes, line sizes, associativity, prefetch behavior and write-allocate and write-back policies just to name a few. For simplicity, we always assumed



(a) (b)

Fig. 6. D-cache miss ratios vs line size

only L1 split instruction/data caches of equal sizes. The primary cache size of interest is 8k (2x4k instruction and data caches) - a realistic scenario for the embedded domain. We did not evaluate the influence of the replacement policy since it has been found that LRU and FIFO outperform the random approach, however do not show significant differences among each other. In all of the experiments reported hereafter the LRU replacement is used. We would like to emphasize that this is only due to the specific behavior of the targeted benchmarks and does not form any restriction for the proposed approach. It is very likely that different applications may greatly benefit from replacement policy changes. The first well expected clear difference in performance was found when the cache line size was changed.

Figures 5 and 6 depict how the cache line size influences the miss ratio. The instruction cache miss ratio is shown in Figure 5, while Figure 6 (a) and (b) demonstrate the miss ratio variation for the data cache. In this experiment, direct mapped cache with instruction and data cache sizes equal to 4k where considered. While for the instruction cache the miss ratio keeps decreasing with increasingly larger cache line sizes, the data cache miss ratio shows a clear optimum at certain sizes. For example the *djpeg* and *cjpeg* curves have a minimum at 16 and 32 byte line sizes. The *adpcm* encoder and decoder perform optimally with 8 and respectively 16 bytes long cache lines. Two benchmarks show slightly deviating behavior - the *epic* and the *mpeg*. Both *mpeg* variants, the encode and the decode, show increasing miss ratios when the line size grows from 8, through, 16, 32, 64 and up to 128 bytes. *Epic* however shows even more surprising properties - while the encode direction shows clear miss ratio minimum in the miss ratio for cache line size of 32 bytes, the decoding part of the benchmark shows a minimum only at 128 byte cache line. This fact, however not shown in the figure was found by performing experiments with 256 byte cache lines. The remaining two benchmarks-*gsm* and *g721* do not show significant changes in our experiment, mainly due to the usage of the C *register* keyword that will assign most of the variables to internal registers. It is interesting to note, however that the instruction cache behavior for the *g721* encoder and decoder shows heavy dependence on the cache line size. To summarize, the optimal flux cache configuration needed for *djpeg* and *cjpeg* should be 4k/32 (4k cache organized into 32 byte lines) for instructions and 4k/32 for data for optimal cache performance. Please note that we did ignore some minor differences in cache miss ratios for 32 byte (0.0257) and 128 byte (0.0157) cases otherwise we would be selecting 4k/128 configuration for the *cjpeg* instruction cache. The same configuration (4k/32) works best for the *epic* encoder, while before starting the *epic* decoder the flux cache is to be ”redesigned” into a 4k/128 considering optimal data cache performance.

5 Conclusions and Future Work

In this paper, we introduced the concept of flux caches and have indicated their performance potential for applications with streaming data access patterns such as multimedia. More precisely, we studied different cache sizes and showed the improvement potential inherent to the studied applications in respect to the line size in the case of 8k cache. Since cache miss ratios do only give an indication about the flux cache performance, currently we are implementing the flux caches on the MOLEN Virtex-II Pro prototype and will report the measured numbers in the near future. In addition, the energy performance analysis of the proposed organization needs careful investigation, together with issues like: data consistency and multiprogramming environment.

References

1. Dejuan, E., Casals, O., Labarta, J.: Cache memory with hybrid mapping. In: 7th International Conference on Modelling, Identification and Control, Grindelwald (1987) 27–30
2. Dejuan, E., Casals, O., Labarta, J.: Management algorithms for an hybrid mapping cache memory. In: International Conference on Mini an Microcomputers and their applications, Sant Feliu (1988) 368–372
3. Jouppi, N.P.: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: ISCA. (1990) 364–373
4. Agarwal, A., Pudar, S.D.: Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In: ISCA. (1993) 179–190
5. Seznec, A.: A case for two-way skewed-associative caches. In: ISCA. (1993) 169–178
6. Chan, K.K., Hay, C.C., Keller, J.R., Kurpanek, G.P., Schumacher, F.X., Zheng, J.: Design of the HP PA 7200 CPU processor chip. Hewlett-Packard Journal **47** (1996) 25–33
7. Milutinovic, V., Markovic, B., Tomasevic, M., Tremblay, M.: The split temporal/spatial cache: Initial performance analysis. Proceedings of SCIzzL-5 (1996) 63–69
8. Sánchez, F.J., González, A., Valero, M.: Software management of selective and dual data caches. In: Technical Committee on Computer Architecture (TCCA) Newsletter. (1997)
9. Ranganathan, P., Adve, S.V., Jouppi, N.P.: Reconfigurable caches and their application to media processing. In: ISCA. (2000) 214–224
10. Zhang, C., Vahid, F., Najjar, W.A.: Energy benefits of a configurable line size cache for embedded systems. In: ISVLSI. (2003) 87–91
11. Hartenstein, R.W., Kress, R., Reining, H.: A new FPGA Architecture for Word-Oriented Datapaths. In: 4th International Workshop on Field Programmable Logic and Applications:Architectures, Synthesis and Applications. (1994) 144–155
12. Trimberger, S.M.: Reprogramable Instruction Set Accelerator. U.S. Patent No. 5,737,631 (1998)

13. Vassiliadis, S., Wong, S., Cotofana, S.: The MOLEN ρμ-Coded Processor. In: 11th International Conference on Field Programmable Logic and Applications (FPL). Volume 2147., Belfast, UK, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2001) 275–285
14. Gordon-Ross, A., Vahid, F., Dutt, N.: Automatic tuning of two-level caches to embedded applications. In: DATE. (2004) 208–213
15. Vassiliadis, S., Wong, S., Gaydadjiev, G.N., Bertels, K., Kuzmanov, G., Panainte, E.M.: The molen polymorphic processor. IEEE Transactions on Computers (2004) 1363– 1375
16. Vassiliadis, S., Gaydadjiev, G.N., Bertels, K., Panainte, E.M.: The molen programming paradigm. In: Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation. (2003) 1–10
17. Kurpanek, G., Chan, K., Zheng, J., DeLano, E., Bryg, W.: Pa7200: A pa-risc processor with integrated high performance mp bus interface. In: COMPCON. (1994) 375–382
18. Veidenbaum, A.V., Tang, W., Gupta, R., Nicolau, A., Ji, X.: Adapting cache line size to application behavior. In: ICS ’99: Proceedings of the 13th international conference on Supercomputing, New York, NY, USA, ACM Press (1999) 145–154
19. Kuzmanov, G., Gaydadjiev, G.N., Vassiliadis, S.: Visual data rectangular memory. In: Proceedings of the 10th International Euro-Par Conference (Euro-Par 2004). (2004) 760–767
20. Edler, J., Hill, M.D.: Dinero IV trace-driven uniprocessor cache simulator. (1998) http://www.cs.wisc.edu/˜markhill/DineroIV.
21. Smith, A.: Cache Memories. Computing Surveys **14** (1982) 473–530
22. Burger, D., Austin, T.M., Bennett, S.: Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308 (1996)
23. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In: 30th Annual International Symposium on Microarchitecture, MICRO30. (1997) 330–335

First-Level Instruction Cache Design for Reducing Dynamic Energy Consumption*

Cheol Hong Kim[1], Sunghoon Shim[1], Jong Wook Kwak[1], Sung Woo Chung[2], and Chu Shik Jhon[1]

[1] Department of Electrical Engineering and Computer Science,
Seoul National University, Shilim-dong, Kwanak-gu, Seoul, Korea
{kimch, shshim, leoniss, csjhon}@panda.snu.ac.kr
[2] Processor Architecture Lab., Samsung Electronics,
Giheung-eup, Gyeonggi-do, Korea
s.w.chung@samsung.com

Abstract. Microarchitects should consider energy consumption, together with performance, when designing instruction cache architecture, especially in embedded processors. This paper proposes a power-aware instruction cache architecture, named Partitioned Instruction Cache (PI-Cache), to reduce dynamic energy consumption in the instruction cache. The proposed PI-Cache is composed of several small sub-caches. When the PI-Cache is accessed, only one sub-cache is accessed by utilizing the locality of applications. In the meantime, the other sub-caches are not accessed, resulting in dynamic energy reduction. The PI-Cache also reduces energy consumption by eliminating energy consumed in tag matching. Moreover, performance loss is little, considering the physical cache access time. We evaluated the energy efficiency by running cycle accurate simulator, SimpleScalar, with power parameters obtained from CACTI. Simulation results show that the PI-Cache reduces dynamic energy consumption by 42% – 59%.

1 Introduction

Energy consumption has become an important design consideration, together with performance, when designing embedded processors. It can be attributed to the limitation on battery capacity and significant thermal problems causing high cooling costs. Unfortunately, as the system performance continues to improve, energy consumption in a processor dramatically increases. Therefore, many researches have focused on the energy efficiency of cache memories to reduce energy consumption in a processor, because caches may consume up to half of total processor energy [1]. Filter cache trades performance for power consumption by filtering power-costly regular cache accesses through an extremely small cache [2]. Bellas et al. proposed a technique using an additional mini cache located between the first-level instruction cache and the CPU core, which reduces signal switching activity and dissipated energy with the help of compiler [3]. Selective-way cache provides the ability to disable a set of the ways in a set-associative

* This work was supported by the Brain Korea 21 Project.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 103–111, 2005.

cache during periods of modest cache activity to reduce energy consumption, while the full cache may remain operational for more cache-intensive periods [4]. Way predicting set-associative caches access one tag and data array initially based on their prediction mechanism, and access the other arrays only when the initial access did not result in a match, which leads to less energy consumption at the expense of longer access time [5].

Energy consumption in the first-level instruction cache (iL1) accounts for a significant portion of total processor energy consumption, because accesses to the iL1 occur almost every cycle [6]. Therefore, total energy consumption in a processor is highly dependent on the energy efficiency of the iL1. In this paper, we propose Partitioned Instruction Cache (PI-Cache) to reduce per-access energy consumption of the iL1 by partitioning it to several sub-caches. We propose the PI-Cache to reduce per-access energy consumption of the iL1 by utilizing high locality of applications in the iL1. It is commonly regarded that the program control flows sequentially. Generally, branch instructions occupy only 0% – 30% of total instructions [7][8]. Therefore, most instructions are expected to be accessed sequentially in the iL1. We try to exploit benefits from this high locality of applications in the iL1. The proposed PI-Cache is not applicable to data caches, since the locality in data caches is inferior to that in instruction caches.

There have been several studies in partitioning the iL1 to several sub-caches. One study closely related to ours is that on the partitioned cache architecture, proposed by Kim et al. [9]. They split the iL1 into several sub-caches to reduce per-access energy cost. Each sub-cache in their scheme may contain multiple pages. In contrast, each sub-cache in our scheme is dedicated to only one page, resulting in more reduction of per-access energy compared to their work by eliminating tag lookup and comparison within the iL1. A cache design to reduce tag area cost by partitioning the cache has been proposed by Chang et al. [10]. They divide the cache into a set of partitions, and each partition is dedicated to a small number of pages in the TLB to reduce tag area cost in the iL1. However, they did not consider energy consumption. When the cache is accessed, all partitions are concurrently accessed in their work.

The rest of this paper is organized as follows. Section 2 and Section 3 present the traditional cache architecture and the proposed cache architecture, respectively. Section 4 discusses our evaluation methodology and shows detailed evaluation results. Section 5 concludes this paper.

2 Traditional Instruction Cache

The traditional instruction cache architecture, including instruction TLB, is shown in Fig. 1. Instruction cache structure focused in this paper is Virtually-Indexed, Physically-Tagged (VI-PT) cache. VI-PT cache is used in many current processors to remove the TLB access from critical path. Virtual address is used to index the iL1 and the TLB is concurrently looked up to obtain physical address. After that, the tag from the physical address is compared with the corresponding tag bits from each block to find the block actually requested. The two-level TLB, a very common technique in embedded processors (e.g. ARM11 [11]), consists of micro TLB and main TLB. Micro TLB is placed over the main TLB for filtering accesses to the main TLB for low power consumption. When a miss occurs in the micro TLB, additional cycle is required to access the main

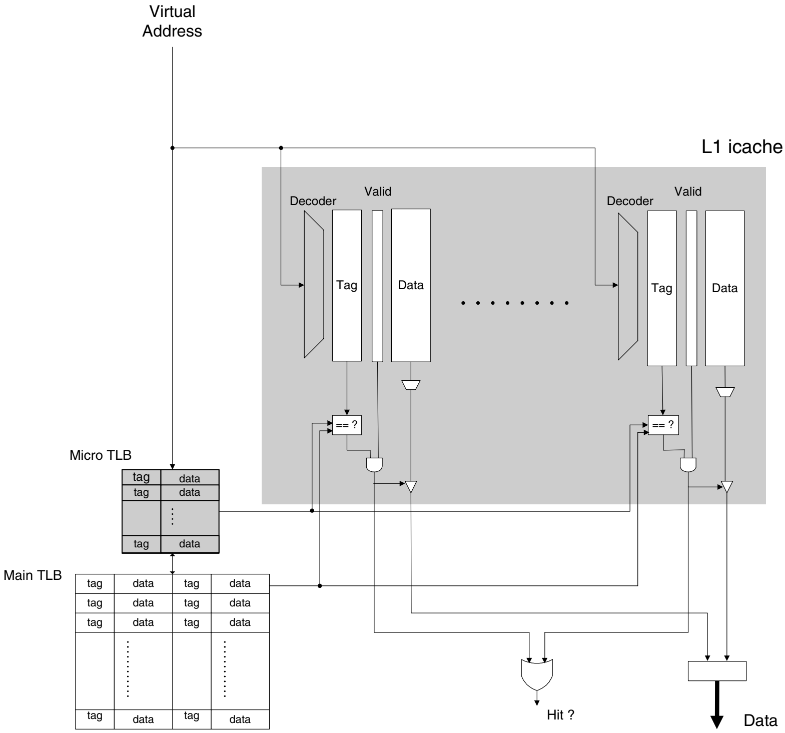


Fig. 1. Traditional Instruction Cache Architecture

TLB. Traditional cache mainly consists of two arrays: tag and actual data. Tag array has the address of actual data. The address tags of cache blocks with same index are compared with the address obtained from the TLB to find the block actually requested. Valid bit identifies the validity of each cache line.

When an instruction fetch request from the processor comes into the iL1, virtual address is used to determine the set. If the selected blocks are not valid, a cache miss occurs. If there is a valid block, the tag of the block is compared with the address obtained from the TLB to see whether it was really requested. If they match, the cache access is a hit.

3 Partitioned Instruction Cache

Energy consumption in the cache is mainly dependent on the cache configuration such as cache size and associativity. In general, small cache consumes less energy than large cache. However, small cache increases cache miss rates, which leads to performance degradation. Thus, large cache is inevitable for performance. The proposed PI-Cache is composed of several small sub-caches in order to make use of both advantages from small cache and large cache. When an access comes into the PI-Cache, only one sub-

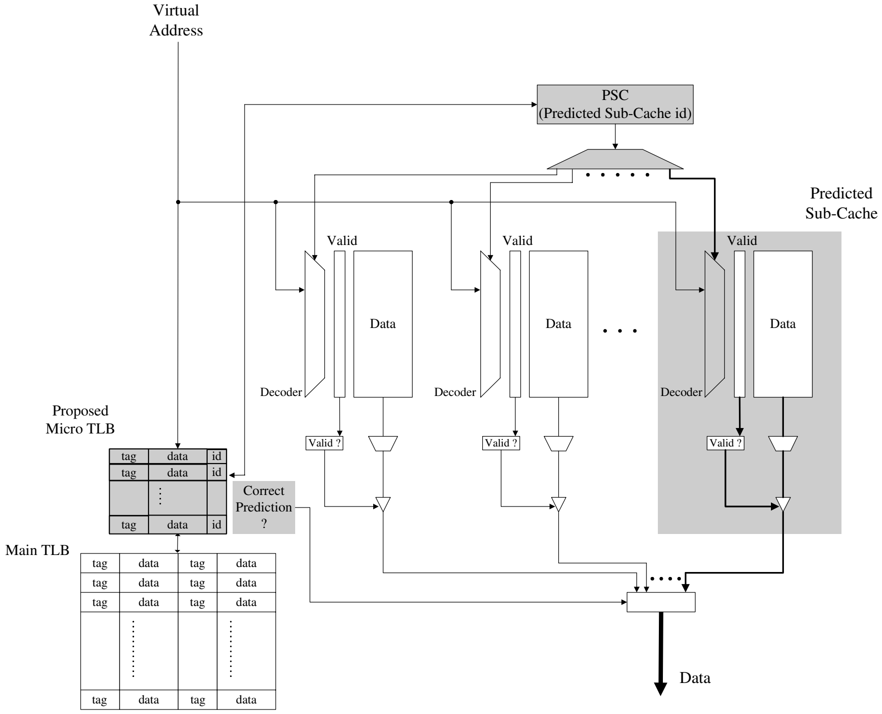


Fig. 2. Proposed PI-Cache Architecture

cache that is predicted to have the requested data, is accessed, and the other sub-caches are not accessed. In the PI-Cache, each sub-cache is dedicated to only one page allocated in the micro TLB. The number of sub-caches in the PI-Cache is equal to the number of entries in the micro TLB. Therefore, there is one-to-one correspondence between sub-caches and micro TLB entries.

Increasing the associativity of the cache to improve the hit rates has negative effects on the energy efficiency, because set-associative caches consume more energy than direct-mapped caches by reading data from all the lines that have same index. In the PI-Cache, each sub-cache is configured as direct-mapped cache to improve the energy efficiency. Each sub-cache size is equal to page size. Therefore, we can eliminate tag array in each sub-cache because all the blocks within one page are mapped to only one sub-cache.

Fig. 2 depicts the proposed PI-Cache architecture. There are three major changes compared with the traditional cache architecture. 1) Id field is added to each micro TLB entry to denote the sub-cache which corresponds to each micro TLB entry. The id field in each micro TLB indicates the sub-cache which all cache blocks within the page are mapped to. 2) There is a register called PSC (Predicted Sub-Cache id) that stores the id of the latest accessed sub-cache. An access to the PI-Cache is performed based on the information in the PSC register. 3) Tag arrays are eliminated in the iL1.

When an instruction fetch request from the processor comes into the iL1, only one sub-cache, which was accessed just before, is accessed based on the information stored in the PSC register. At the same time, the access to the instruction TLB is performed. If the access to the micro TLB is a hit, it means that the requested data is within the pages mapped to the iL1.

In case of a hit in the micro TLB, the id of the matched micro TLB entry is compared with the value in the PSC register to verify the prediction. When the prediction is correct (the sub-cache id corresponding to the matched micro TLB entry is same to that stored in the PSC register), a normal cache hit occurs if data was found in the predicted sub-cache and a normal cache miss occurs if data was not found. A normal cache hit and a normal cache miss mean a cache hit and a cache miss without penalty (another sub-cache access delay), respectively. If the prediction is not correct, it means that the requested data belongs to the other pages in the iL1. In this case, the correct sub-cache is also accessed. This incurs additional cache access penalty. If data is found in the correct sub-cache, a cache hit with penalty occurs. If cache miss occurs even in the correct sub-cache, a cache miss with penalty occurs.

If a miss occurs in the micro TLB, it implies that the requested data is not within the pages mapped to the iL1, consequently a cache miss occurs. In this case, the sub-cache that corresponds to the replaced page from the micro TLB is flushed in whole. Each sub-cache can be easily flushed by resetting valid bits of all cache blocks because the iL1 only allows read operation (No write-back is required). Then, incoming cache blocks which correspond to the newly allocated entry in the micro TLB are placed into the flushed sub-cache.

Table 1. Memory Hierarchy Parameters

Parameter	Value
Micro TLB	fully associative, 1 cycle latency
Main TLB	32 entries, fully associative, 1 cycle latency, 30 cycle miss penalty
L1 I-Cache	16KB and 32KB, 1-way – 8-way, 32 byte lines, 1 cycle latency
Sub-cache in the PI-Cache	4KB (Page size), 1-way, 32 byte lines, 1 cycle latency
L1 D-Cache	32KB, 4-way, 32 byte lines, 1 cycle latency, write-back
L2 Cache	256KB unified, 4-way, 64 byte lines, 8 cycle latency, write-back
Memory	64 cycle latency

In the PI-Cache, there is no conflict miss, since one page is mapped to one sub-cache whose size is equal to page size. However, there are more compulsory misses than the traditional cache, since the sub-cache in the PI-Cache is flushed whenever the corresponding entry in the micro TLB is replaced. Compulsory misses may be eliminated if the PI-Cache transfers all cache blocks in the page simultaneously when the page is allocated in the micro TLB. However, the PI-Cache does not transfer whole page at the same time, because it may incur serious bus contention problem. The PI-Cache transfers the cache block from lower level memory only when it is requested.

The PI-Cache incurs little hardware overhead. Traditional micro TLB must be extended to incorporate the id for each entry for this scheme. However, the number of bits for id field is typically small: 2, 3 or 4 bits. One register is required for the PSC register and one comparator is required to check sub-cache prediction. This overhead is negligible.

The PI-Cache is expected to reduce dynamic energy consumption by reducing the size of accessed cache and eliminating tag comparison. If the hit rates in the PI-Cache do not decrease so much compared to those in the traditional cache, the PI-Cache can be an energy-efficient alternative as an iL1.

4 Experiments

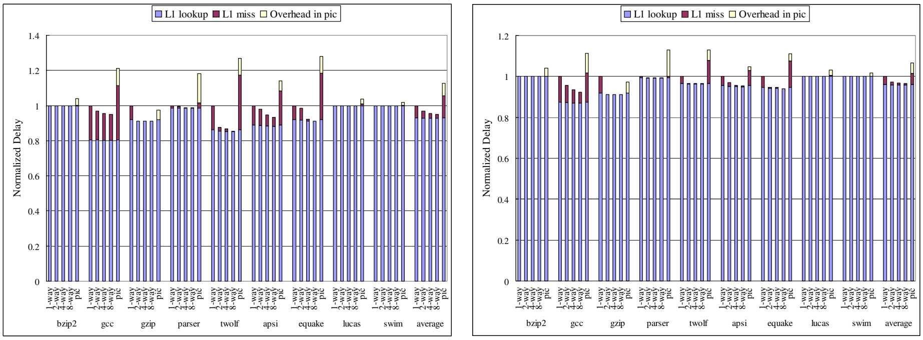
In order to determine the characteristics of the proposed PI-Cache with respect to the traditional caches, we simulated various benchmarks using SimpleScalar simulator [12]. CACTI cache energy model was used to collect the power parameters where we assumed 0.18um technology [13]. Simulated applications are selected from SPEC CPU2000 suite [8]. Memory hierarchy parameters used in this simulation are shown in Table 1. The simulated processor is a 2-way superscalar processor with an L2 cache, which is expected to be similar to the next generation embedded processor by ARM [14].

4.1 Cache Delay

The normalized instruction fetching delay obtained from simulations is given in Fig. 3. We assume that *pic* in the graphs denotes the proposed PI-Cache. *L*1 *lookup* portion in the bar represents the cycles required for iL1 accesses. *L*1 *miss* portion denotes the cycles incurred by iL1 misses. *Overhead in pic* portion denotes the delayed cycles incurred by sub-cache misprediction in the PI-Cache scheme, namely the penalty to access another sub-cache after misprediction. Note that traditional cache schemes do not have *Overhead in pic* portion.

As shown in these graphs, set-associative caches show less cache delay than direct-mapped caches by reducing the delay due to cache misses. This comes from the fact that set-associative caches improve the hit rates compared to direct-mapped caches by reducing conflict misses in the cache. Therefore, the cache delay is reduced with more degree of associativity. As shown in Fig. 3, the performance of 16KB PI-Cache is degraded by 12% on average compared to that of the traditional direct-mapped cache. 32KB PI-Cache is degraded by 6% on average. This performance degradation is caused by two reasons: one is the degradation of the hit rates by restricting the blocks to be allocated in the iL1 to the blocks within the pages mapped to the micro TLB entries. The other is the sub-cache misprediction which incurs additional sub-cache access delay, indicated by the *Overhead in pic* portion.

Performance gap between the traditional caches and the PI-Cache decreases as the cache size increases. This is because the hit rates in the PI-Cache improve by increasing the number of the pages mapped to the iL1: 32KB PI-Cache with 8 pages (sub-caches) compared with 16KB PI-Cache with 4 pages (sub-caches). As shown in Fig. 3, *L*1 *miss* portion in the bars significantly decreases in 32KB PI-Cache compared to 16KB PI-



(a) 16KB L1 I-Cache

(b) 32KB L1 I-Cache

Fig. 3. Normalized Cache Delay

Table 2. Physical Cache Access Time

Access Time, ns	1-way	2-way	4-way	8-way	PI-Cache
16KB	1.129	1.312	1.316	1.383	0.987
32KB	1.312	1.455	1.430	1.457	0.987

Cache. From these results, the PI-Cache is expected to be more efficient as the cache size becomes larger.

The results shown in Fig. 3 are obtained from the configurations in Table 1. We simulated all cache configurations with same cache access latency (1 cycle). In fact, physical access time varies by the cache configurations. Table 2 gives the physical access time according to each cache models obtained from CACTI model. For the traditional cache models, physical cache access time generally increases as the degree of the associativity increases. Physical access time of the PI-Cache is “1 AND gate delay (it is required to enable the sub-cache indicated by the PSC register, 0.114 ns, obtained from ASIC STD130 DATABOOK by Samsung Electronics [15]) + Sub-cache access latency (0.873 ns, obtained from CACTI)”. As shown in Table 2, physical access time of the PI-Cache is faster than the direct-mapped traditional cache, because accessed cache size is small and tag comparison is eliminated in the PI-Cache. This feature is well shown in 32KB iL1 than 16KB iL1. Physical access time of the traditional cache increases if the cache size increases. However, physical access time for the PI-Cache is dependent on the sub-cache size, not on the cache size. Consequently, access time of the PI-Cache is independent of the cache size. Therefore, if the processor clock speeds up or the size of cache increases in the future, the proposed PI-Cache is expected to be more favorable.

4.2 Energy Consumption

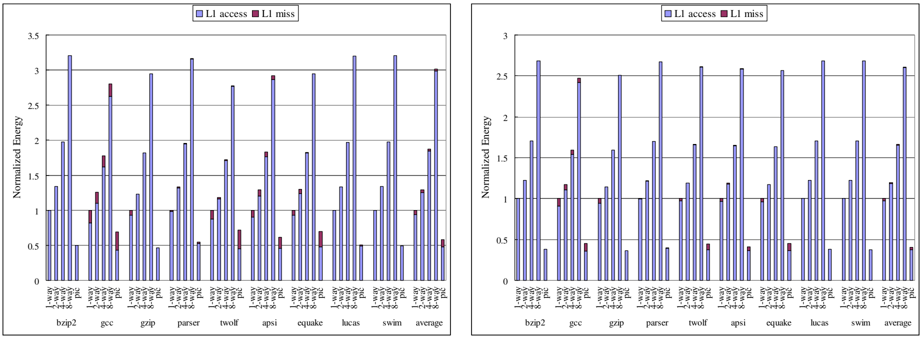
Table 3 shows per-access energy consumption according to each cache models obtained from CACTI model. In the traditional caches, the energy consumed by the cache increases as the degree of associativity increases. The increase in associativity implies

Table 3. Per-access Energy Consumption

Energy, nJ	1-way	2-way	4-way	8-way	PI-Cache
16KB	0.473	0.634	0.935	1.516	0.232
32KB	0.621	0.759	1.059	1.666	0.232

the increase of output drivers, comparators, sense amplifiers, consequently the increase of total energy. The PI-Cache consumes less per-access energy compared to traditional caches. There are two reasons for better energy efficiency: one is that the size of cache accessed in the PI-Cache is smaller than the traditional cache by partitioning it to several sub-caches. The other is the elimination of accesses to tag arrays in the PI-Cache.

As shown in Table 3, per-access energy in the traditional caches increases as the cache size increases. By contrast, per-access energy in the PI-Cache is independent of the cache size, since per-access energy in the PI-Cache is dependent on the sub-cache size that is equal to the page size. This size-independent property of energy consumption is especially favorable in a large cache.



(a) 16KB L1 I-Cache

(b) 32KB L1 I-Cache

Fig. 4. Normalized Cache Energy Consumption

Detailed energy consumption obtained from SimpleScalar and CACTI together is shown in Fig. 4. Total energy consumption presented in these graphs is the sum of the dynamic energy consumed during instruction fetching. In Fig. 4, *L*1 *access* portion denotes the dynamic energy consumed during iL1 accesses, and *L*1 *miss* portion represents the dynamic energy consumed during accessing lower level memory incurred by misses in the iL1. 16KB PI-Cache gives the improvement of energy efficiency by 42% on average and 32KB PI-Cache reduces energy consumption by 59% on average. As expected, the PI-Cache is more energy efficient with a large cache.

5 Conclusions

We have introduced a new instruction cache design called Partitioned Instruction Cache (PI-Cache) to reduce energy consumption in embedded processors. The PI-Cache is composed of several sub-caches and each sub-cache is dedicated to only one page. When an access from the processor core comes into the PI-Cache, only one predicted sub-cache is accessed for improving energy efficiency. The proposed PI-Cache reduces energy consumption significantly compared to traditional caches. This energy efficiency in the PI-Cache comes in two ways: one is reducing the size of cache accessed and the other is eliminating tag comparison. Moreover, performance loss is little, considering the reduced physical cache access time. Therefore, the PI-Cache is expected to be a scalable solution for a large instruction cache.

References

1. Segars, S.: Low power design techniques for microprocessors. In: Proceedings of International Solid-State Circuits Conference. (2001)
2. Kin, J., Gupta, M., Mangione-Smith, W.: The filter cache: An energy efficient memory structure. In: Proceedings of International Symposium on Microarchitecture. (1997) 184-193
3. Bellas, N., Hajj, I., Polychronopoulos, C.: Using dynamic cache management techniques to reduce energy in a high-performance processor. In: Proceedings of International Symposium on Low Power Electronics and Design. (1999) 64–69
4. Albonesi, D.H.: Selective cache ways: On-demand cache resource allocation. In: Proceedings of International Symposium on Microarchitecture. (1999) 70–75
5. Powell, M., Agarwal, A., Vijaykumar, T.N., Falsafi, B., Roy, K.: Reducing set-associative cache energy via way-prediction and selective direct-mapping. In: Proceedings of International Symposium on Microarchitecture. (2001) 54–65
6. Montanaro, J., et al.: A 160 Mhz, 32b, 0.5W CMOS RISC microprocessor. In: Proceedings of International Solid-State Circuits Conference. (1996) 214–229
7. Lee, C., Potkonjak, M., Mangione-Smith, W.: A tool for evaluating and synthesizing multimedia and communications systems. In: Proceedings of the 30th Annual International Symposium on Microarchitecture. (1997) 330–335
8. SPEC CPU2000 Benchmarks. http://www.specbench.org
9. Kim, S., Vijaykrishnan, N., Kandemir, M., Sivasubramaniam, A., Irwin, M.J.: Partitioned instruction cache architecture for energy efficiency. ACM Transactions on Embedded Computing Systems **2** (2003) 163–185
10. Chang, Y.-J., Lai, F., Ruan, S.-J.: Cache design for eliminating the address translation bottleneck and reducing the tag area cost. In: Proceedings of International Conference on Computer Design. (2002) 334
11. ARM Corp.: ARM1136J(F)-S. available at http://www.arm.com/products/CPUs/ARM1136JF-S.html
12. Burger, D., Austin T.M., Bennett, S.: Evaluating future micro-processors: The SimpleScalar tool set. Technical Report TR-1308, Univ. of Wisconsin-Madison Computer Sciences Dept. (1997)
13. Shivakumar, P., Jouppi, N.P.: CACTI 3.0: An integrated cache timing, power, and area model. TR-WRL-2001-2 (2001)
14. Muller, M.: At the Heart of Innovation. available at http://www.arm.com/miscPDFs/6871.pdf
15. Samsung Electronics: ASIC STD130 DATABOOK. (2001)

A Novel JAVA Processor for Embedded Devices

Yiyu Tan, Chihang Yau, Kaiman Lo, Paklun Mok, and Anthony S. Fong

Department of Electronic Engineering, City University of Hong Kong,
Tat Chee Avenue, Kowloon Tong, Hong Kong
anthony.fong@cityu.edu.hk

Abstract. As a result of its object-oriented (OO) feature and corresponding advantages of security, robustness and platform independence, Java is widely applied in embedded devices. However, among current solutions to Java execution engine implemented by software or hardware, the overheads of executing OO related bytecodes are costly and have a great impacts on the overall performance of Java applications, especially in embedded devices, where real-time operations and low power consumptions are required in the case of limited memory. To solve this problem, a novel Java processor architecture called jHISC is proposed where the OO related bytecodes are supported in hardware directly. In jHISC, an object is represented by the hardware-readable data structure -object context, which then makes it possible to implement complex OO related bytecodes at hardware level and access some fields of object in parallel to improve the execution speed. It mainly targets J2ME and implements about 93% bytecodes and 83% OO related bytecodes in hardware directly, and the OO related operations are executed much faster in jHISC than by software traps.

1 Introduction

JAVA was introduced in the mid-1990s to overcome the major weakness of C and C++ and is now widely applied in network applications and embedded devices, such as PDAs, mobile phones, TV set-up boxes and Palm PCs [1]. A new report from ARC Group estimated the number of J2ME(Java 2 Micro Edition) compatible handsets was 421 millions in 2003, 442 millions in 2004, and 1 billion in 2006 [2]. Java claims to be more robust, secure and portable in addition to its inheriting the common advantages of object-oriented programming language such as encapsulation, polymorphism, dynamic binding and inheritance. Its increasing robustness and security can be attributed to the automatic garbage collection, static and runtime type checking, exception handling mechanism, array boundary checking and restrictive object reference management [3] while its enhanced portability is realized through compilation and execution of Java machine instructions called bytecodes instead of particular processor binaries.

Amongst the three traditional ways of executing Java bytecodes, interpreter, the original method, finds its performance significantly affected by the time-consuming loops in the course of software emulation, though boasting its simple interpretation, relatively easy implementation, and little memory demand. The second way Just-In-Time (JIT) compiler, however, also introduces additional compilation overheads and requires much more memory in spite of its advantages over interpreter in eliminating

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 112–121, 2005.

redundant translations and optimizing generated native instructions during compilation. In view of disadvantages of the above two methods, Java processor, executes Java bytecodes directly through implementation of Java Virtual Machine (JVM) by hardware. In addition, tailoring hardware support for some Java special features such as security, multi-threading and garbage collection can potentially enable Java processor to deliver much better performance than a general-purpose processor for Java applications. Compared with the other approaches, Java processor appears to be particularly suitable for embedded devices.

Accordingly, high performance Java processors have been developed by many companies and researchers in recent years [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. In this paper, we propose jHISC, a novel architecture Java processor based on J2ME and mainly targeting embedded devices. The rest of this paper summarizes major previous researches on Java processors in section 2, and describes jHISC architecture in section 3. In section 4, the system implementation results and execution performance of some main OO related bytecodes are presented. Finally, conclusions are made in section 5.

2 Related Work

Among the proposed solutions to Java processors in recent years, the method to support bytecodes can be mainly summarized into three categories: replacing JVM by a hardware stack machine, hardware translation, and coprocessor. Each of them has its advantages and disadvantages.

Replacing JVM by a hardware stack machine is the most popular and easiest way because JVM is basically a software stack machine. Processors of this type use bytecodes as their native instructions and execute them directly, such as PicoJava I and II from Sun Microsystems, aJ-100 from aJile Systems, Inc. [4, 5, 6, 7]. However, the performance and efficiency of stack machine are quite low since all operands such as temporary data, intermediate values, and method arguments are pushed into or popped from stack during execution, which will add overheads. It is also difficult for processors to handle complex operations due to a lot of data required to pop or push from stack, such as OO related operations, which leads them to need the assistances of software traps or microcode. Additionally, since they are pure Java processors, it is inflexible for them to execute the application programs written by other programming languages if there are no compilers supporting.

For Java processors implemented by hardware translation method, a small translator is added between instruction fetch and decoding units in a general-purpose processor core to convert most simple bytecodes to native instructions by one to one or N to one at run-time. For the other complex bytecodes, such as OO related bytecodes, it invokes software traps to perform them. ARM Jazelle and JA108 are two commercial products by using this method [8, 9]. Some researchers also adopted this technique, for example, R. Radhakrishnan etc. and M. Schoeberl accelerated Java performance by hardware interpretation [11, 17]; J. Glossner and S. Vassiliadis developed Delft-Java by directly translating most bytecodes into Delft-Java instructions [15, 16]. The cost of hardware translation is relatively low because only a small circuitry is added and little impact is introduced due to its transparence to the host. Furthermore, the processor executes the application programs written by other programming languages that the host architecture supports. But some features of Java language such as security and object-oriented programming features may be compromised if the host architecture does not support them at hardware level.

Coprocessor approach simply attaches a Java coprocessor to a general-purpose processor, thus Java bytecodes and non-Java bytecodes are performed by coprocessor and general-purpose processor, respectively, to guarantee the system to execute both Java and traditional application programs written by other programming languages directly. AU-J2000 from Aurora VLSI Inc. is a 32-bit dual-processor where a Java coprocessor based on hardware stack machine is combined with a general-purpose processor core [10]. Several researchers also implemented coprocessors by reconfigurable systems independent outside the general-purpose processors to accelerate the Java executions [12, 13, 14]. The coprocessor provides a good support to Java without affecting the compatibility of core. But chip area and power consumption increase significantly.

However, almost all the current Java processors do not support object-oriented programming at architectural level so that they perform OO related operations by software traps where an OO related operation may cost several ten clock cycles, sometimes more than one hundred clock cycles [6]. Moreover, the OO related operations constitute about 15% of all operations [18, 19], their executions thus have significant impacts on the execution speed of Java programs. Although in some solutions, a quick version replacement scheme of OO related bytecodes was adopted to speed up execution, it also increased the chip area and power consumption because the quick version of the related bytecode was performed by microcode which implementation needed a lot of ROMs [4, 6]. Their performance penalty does not fit well with the requirements of embedded devices, such as real-time operations, low power consumptions in the case of limited memory. In addition, many application programs written by other programming languages are now available, which makes it desirable to have a general-purpose processor with enhanced architectural features to support object-oriented programming in hardware directly. All these lead us to develop jHISC, a novel architecture Java processor supporting OO related operations at the hardware level.

3 jHISC Architecture

jHISC is a 32-bit object-oriented processor based on High Level Instruction Set Computer (HISC) architecture [20, 21, 22]. It supports the object-oriented programming and access control in hardware directly by modifying the descriptor structure. By letting hardware know what an object is, processor provides various support to manage the system, such as object management, memory management etc. jHISC mainly targets such J2ME applications as smart mobile phones, PDAs and the other embedded devices, but the float-point and 64-bit operations are not supported in current version.

3.1 Object Model

The object representation method is critical in object-oriented programming systems due to its affects on the speed of accessing objects. In jHISC, an object is represented by the hardware-readable data structure-object context, which consists of object header, data space and corresponding descriptor tables, etc. Three kinds of contexts, namely instance, class, and method contexts, are mapped to the hardware architecture and distinguished by the object header. The format of the object header is shown in Fig. 1.

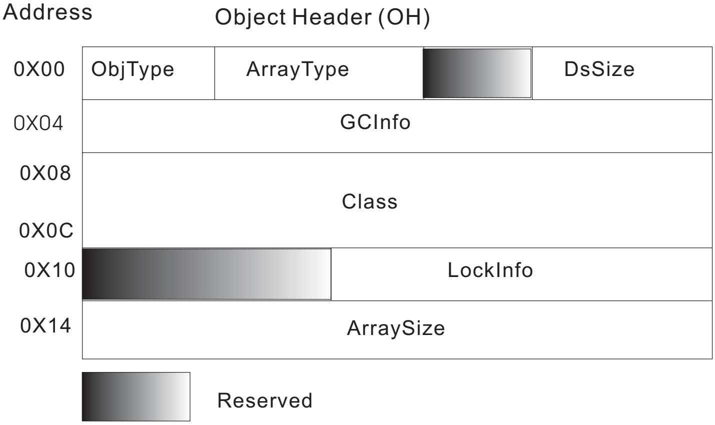


Fig. 1. The format of an object header

Inside an object header, the object type(i.e., instance, class, method and array) is stored in the field *ObjType*; Field *DsSize* specifies the size of the related data space; Field *GCInfo* stores information to give hardware support for real-time garbage collections; A reference pointer is put into field *Class* to link the object with its affiliated class; *ArraySize* and *ArrayType* specify the number and type of the elements in an array, respectively, when the object is an array; *LockInfo* is used for multithreading.

Except the object header, an instance context also includes Instance Header (IH) and Instance Data Space (IDS); a class context consists of Class Header (CH), Class Operand Descriptor Table (CODT), Class Property Descriptor Table (CPDT) and Class Data Space (CDS); and a method context includes Method Header (MH), Method Code Space (MCS) and Local Variable Frame (LVF) for local variable storage. In addition, when applied to represent an array, an instance context also includes array data area. And inside the class context, CODT and CPDT store class operand descriptors and class property descriptors, respectively. Class operand descriptors indicate the resources accessed by the class and class property descriptors assert the properties owned by the class. The different object context structures and their relations are shown in Fig. 2.

Typically, each object has a unique object context according to its type, and a reference always points to the base address of object header after the object is resolved. In the object context, each component is stored with a constant offset to the object header, thus some components can be accessed in parallel to reduce the access overhead.

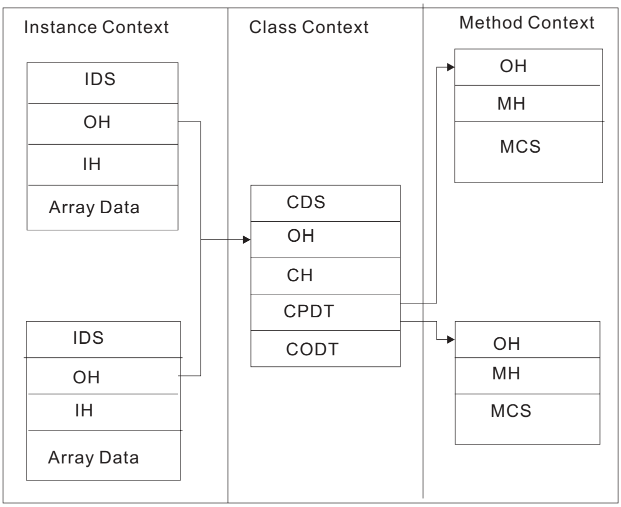


Fig. 2. Different object structures and their relations

3.2 Descriptor Format

In jHISC, 32-bit operand descriptors describe properties owned by a class or resources accessed by this class. Their format is shown in Fig. 3 and similar to that described by the Java specification, including *Address Field, Type Field, Static Flag, Access Modifier, Read-Only Flag, and Resolved Flag*. *Address Field* provides a byte offset to locate data in the data spaces. *Access Modifier* is used for security control, and four access modifiers (public, private, protect and package) are defined in the current system. *TypeField* stores the types of described data defined for both primitive and reference types. *Static flag* indicates where the data are stored. For the non-static field, data are stored inside Instance Data Space (IDS) of the target reference, otherwise, inside Class Data Space (CDS). *Read-only flag* represents whether the target can be written. And *Resolved Flag* indicates whether the reference is resolved or not. If not, the system will be trapped to the operating system routines for the dynamic reference resolution.

Resolved Flag[31]	Read-only Flag [30]	Static Flag [28:29]	Type Field [27:24]	Access Modifier [23:22]	Address Field [21:0]

Fig. 3. Operand descriptor format

3.3 Instruction Set

jHISC is a RISC processor with some enhancements for the OO operations. Its instruction set is compatible with MIPS 32 except the memory-register data transfer and OO related instructions. Memory-register data transfer instructions allow programs to access memory directly in traditional computers, which may result in security problems. In jHISC, all data are encapsulated into objects, and each object associates with a pair of memory boundaries (upper and lower boundary). A program needs to pass the bound control checks before it accesses the data and out-of-boundary accesses are prohibited. And the OO instructions *gifld* and *pifld* are added to perform data transfer operations between memory and register with rigid memory access checks. Additionally, jHISC also provides object and array manipulation instructions to handle the related operations. To improve the execution efficiency, bytecode *invokevirtual* is divided into two instructions, namely *ivkinstance* and *ivkinternal*, according to the invoked instance method. If the invoked instance method is within the same class as the current method, the instruction *ivkinternal* is executed, otherwise, *ivkinstance* instead. The motivation is that the overhead of invoking an instance method within the same class is much smaller since it only needs to switch the involved method contexts. And the similar way is also applied to the bytecode *getfield* and *putfield.*

Table 1. The bytecodes supported by jHISC

Number of bytecodes	226
Number of bytecodes excluding the float-point and 64-bit operations	140
Number of bytecodes supported by the hardware directly	130
Number of bytecodes done by the software traps	10
Number of bytecodes for OO operations	40
Number of OO bytecodes supported by the hardware directly	33
Percentage of bytecodes supported by the hardware directly	93%
Percentage OO bytecodes supported by the hardware directly	83%

Excluding the float-point and 64-bit operation instructions, jHISC implements 93% bytecodes and 83% OO related bytecodes in hardware directly. The rests are executed through software traps, such as *new*, *newarray*, because their executions are very complex and require the assistance of operating system. The corresponding details are shown in Table 1.

Java bytecodes can be converted into jHISC instructions in instruction folding unit by one to one or N to one so as to reduce code density and to improve program execution speed. Moreover, the application programs written by other programming language can be performed on the jHISC platform easily since its instruction set is compatible with MIPS 32.

3.4 System Architecture

Basically, jHISC is RISC architecture, and the block diagram of the whole system is shown in Fig. 4. The system is implemented by 5 pipelines, including instruction fetch, instruction folding and decoding, data fetch, execution and write-back. Compared with the traditional RISC architecture, jHISC adds Instruction Queue, Translation and Data Buffer units. Instruction Queue, Translation unit consists of an instruction buffer, an instruction folding manager and a stage controller. The instruction buffer is made up of eight registers with each storing an instruction and its corresponding program counter. And the instruction folding manager is used to realize the folding algorithm to convert the bytecodes to jHISC instructions. Data buffer unit consists of sixteen multi-port registers so that data can be read or written synchronously to reduce the accessing time.

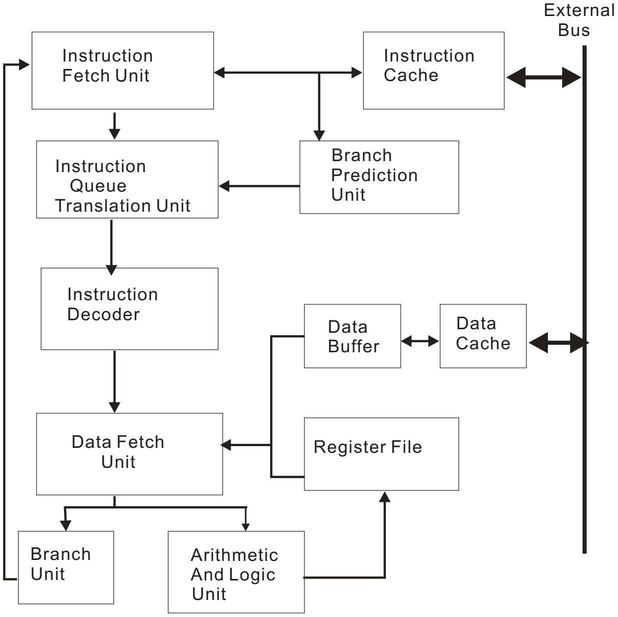


Fig. 4. Block diagram of system architecture

4 System Implementation and Timing

The whole system with 4KB instruction cache and 8KB data cache was described by VHDL and implemented by a Xilinx Virtex FPGA XCV800 to verify our concept and the corresponding chip is currently under development. During implementation, the caches were generated by Xilinx CORE Generator and the whole system cost about 600K equivalent gates in FPGA. Table 2 shows the map results reported by Xilinx ISE 6.0.

Table 2. The bytecodes implemented through software traps in jHISC

Logic Utilization:	
Number of Slice Flip Flops:	3,963 out of 18,816 21%
Number of 4 input LUTs:	13,090 out of 18,816 69%
Total Number 4 input LUTs:	14,743 out of 18,816 78%
Number used as logic:	13,090
Number used as a route-thru:	1,629
Number used as 16x1 ROMs :	24
Number of bonded IOBs :	174 out of 316 55%
Number of Tbufs :	3,424 out of 9,632 35%
Number of Block RAMs :	28 out of 28 100%
Number of GCLKs :	1 out of 4 25%
Number of GCLKIOBs :	1 out of 4 25%
Total equivalent gate count for design:	598,463

Table 3. The number of clock cycles needed by some main OO bytecodes in jHISC and software traps in PicoJava II. The clock cycles consumed by the method revocation instruction *oo_rvk* are 3, 5, 7, respectively, in the case of *ivkinternal*,*ivkclass* and *ivkinstance*. And the value in the table is the average of them

Bytecodes in PicoJavaII II	clock cycles	Instruction in jHISC	clock cycles (all data hit in the cache)	clock cycles (all data miss in the cache)
getfield	107	gfld	6	18
		gifld	2	6
putfield	98	pfld	6	18
		pifld	2	6
getstatic	80	gsfld	6	18
putstatic	79	psfld	6	18
invokestatic	58	ivkclass	9	27
invokevirtual	150	ivkintance	9	27
		ivkinternal	5	15
ireturn	8	oo_rvk	5	7
return	8			
areturn	8			
checkcast	97	checkcast	3	9
instanceof	100	instanceof	4	12

To estimate the execution performance, we counted the number of clock cycles needed by each bytecode execution in jHISC. Since the time cost by instruction executions is not exact in FPGA, the results are mainly based on simulations in jHISC, partly from simulations of RTL level. Similar to the other Java processors, most of the simple bytecodes, such as load, store operations, could be executed from one to three clock cycles in jHISC. However, for object manipulation operations, they were executed much faster in jHISC than by software traps. Table 3 shows the number of clock cycles needed by some main object manipulation bytecodes in jHISC and in PicoJava II where OO related bytecodes were executed by software traps firstly and the number of clock cycles consumed was estimated by simply counting the number of the involved bytecodes in software traps.

5 Conclusion

Embedded devices becomes more and more popular now, and a lot of complex application programs written by Java, such as complex games, network applications, begin to be applied in them, which also put forward rigid requirements to processors in them, for example, high execution efficiency and low power consumption. In the view of these, jHISC offers an attractive solution for embedded devices to speed up Java program executions while enforcing the security of object-oriented programming and program compatibility to the existing systems. Firstly, both the hardware implementation of complex OO related bytecodes and parallel access of object information contribute to the performance improvement since it uses the hardware-readable data structure to represent objects and each field of object is stored in a specific address. Secondly, built-in bounds checking to guarantee no out-of-boundary accessing objects results in the enhancement of security because all information is encapsulated into objects and no operations access memory directly. Thirdly, both RISC-based architecture with enhanced features to support object-oriented programming and instruction set compatible with MIPS 32 make it possible for the application programs written by other programming languages to be performed on jHISC. Additionally, in order to improve execution speed in further, we can add a method cache to store the related reference addresses to reduce the accesses to memory.

Acknowledgements

This work was supported partly by City University of Hong Kong under Strategic Research Grant 7001548.

References

1. Lee, Y.M., Tak, B.C., Maeng, H.S., Kim, S.D.: Real-time java virtual machine for information appliances. IEEE Transactions on Consumer Electronics **46** (2000) 949
2. : (2005) Http://www.anfymobile.com/market/mgaming.html.
3. Grand, M.: Java Language Reference. O'Reilly (1997)
4. O'Connor, J.M., Tremblay, M.: Picojava-i: The java virtual machine in hardware. IEEE MICRO (1997) 45
5. McGhan, H., O'Connor, J.M.: Picojava: A direct execution engine for java bytecode. Computer (1998) 22
6. Sun Microsystems: PicoJava-II: Java Processor Core. (1998)
7. aJile Systems, Inc.: aJ-100 Real-time Low Power JavaTM Processor. (2001)
8. ARM: Jazelle Technology for Java Application. (2001)

9. NAZOMI Communications Inc.: JA108 – Multimedia Application Processor. (2003)
10. Aurora VLSI Inc.: AU-J2000: Super High Performance Java Processor Core. (2000)
11. Radhakrishnan, R., Bhargava, R., John, L.K.: Improving java performance using hardware translation, ACM International Conference on Supercomputing (2001) 427
12. Kent, K.B., Serra, M.: Hardware/software co-design of a java virtual machine, IEEE International Workshop on Rapid Systems Prototyping (2000) 66
13. Lattanzi, E., Gayasen, A., Kandemir, M., et al: Improving java performance using dynamic method migration on fpgas, The 18th International Parallel and Distributed Processing Symposium (2004) 134
14. Ha, Y., Hipik, R., Vernalde, S., Verkest, D.: Adding hardware support to the hotspot virtual machine for domain specific applications. Lecture Notes In Computer Science **2438** (1997) 45
15. Glossner, C.J., Vassiliadis, S.: The delft-java engine: An introduction, The 3th International Euro-Par Conference on Parallel Processing (1997) 766
16. Glossner, C.J., Vassiliadis, S.: Delft-java link translation buffer, The 24th Conference on EuroMicro (1998) 221
17. Schoeberl, M.: Jop: A java optimized processor. Lecture Notes in Computer Science **2889** (2003) 346
18. Vijaykrishnan, N., Ranganathan, N.: Supporting object accesses in a java processor. IEE Proc.-Comput. Digit. Tech. **147** (2000) 435
19. Lun, M.P., Fong, A., Hau, G.K.W.: Object-oriented processor requirements with instruction analysis of java programs. ACM SIGARCH Computer Architecture News **31** (2003) 10
20. Lun, M.P., Li, R., Fong, A.: Method manipulation in an object-oriented processor. ACM SIGARCH Computer Architecture News **31** (2003) 18
21. Fong, A.S.: A computer architecture with access control and cache option tags on individual instruction operands. ACM SIGARCH Computer Architecture News **31** (2003) 1
22. Fong, A.S.: Hisc: A high-level instruction set computer, The 7th European Simulation Symposium (1995) 406

Formal Specification of a Protocol Processor

Tomi Westerlund[1,2] and Juha Plosila[2]

[1] Turku Centre for Computer Science,
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
tomi.westerlund@utu.fi

[2] Department of Information Technology, University of Turku,
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
juha.plosila@utu.fi

Abstract. To ensure the correctness of functional and temporal properties of modern network hardware devices is becoming increasingly challenging because the growing complexity and demanding time-to-market requirements. In this paper we address the problem by deriving a TACO protocol processor model in the formal framework of Timed Action Systems. Formal methods offer a prominent approach to specify, design, and verify such devices with the benefits of a rigorous mathematical basis. The derivation demonstrates the capability of preserving correctness when considering an important hardware design decision.

1 Introduction

Formal methods are emerging as a prominent approach to model timed systems. They provide an environment to specify, design and verify timed systems with the benefits of a rigorous mathematical basis. The Action Systems formalism [1] is a state based formal description language initially proposed by Back and Kurki-Suonio [2]. It is based on an extended version of a guarded command language introduced by Dijkstra [3]. It is used for specification and correctness preserving development of reactive systems. It was first tailored to a software system design but is then successfully applied also to hardware system design, both synchronous [4] and asynchronous [5]. It offers a powerful stepwise design environment for designing embedded hardware-software systems throughout the design project from abstract specification to implementable specification. We are able to formally verify each derivation step within the refinement calculus. The time extended Action Systems formalism is presented in [6].

In this paper we show a formal development of a general TACO protocol processor. A TACO (Tools for Application-specific Hardware/Software Codesign) protocol processor framework [7] provides tools and methods for helping the designer in specifying, simulating, evaluating, and synthesising programmable protocol processors. It contains system-level simulation, physical estimation, and synthesis models for a transport triggered base protocol processor architectures. We start from a conventional sequential program describing the behaviour of the general TACO protocol processor. Through several refinement steps we end up a non-trivial model where functional units operate in parallel. From the formal timing model we obtain information of the relative ordering of the system's components: sockets and functional units. The relative ordering is

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 122–131, 2005.

given in form of time constraints that are used in verification of temporal properties of a system, that is, in ensuring the correct operation of the processor model.

Designing processors using Action Systems is already investigate in [8, 9]. The difference with this paper and the two mentioned one, is that we use the time extended Action Systems. Thus, we are able to model both functional and temporal properties of the system. Furthermore, our target is a synchronous representation of the system, where in [8, 9] designed an asynchronous microprocessors. There exists also other formalisms to design synchronous hardware systems, e.g. in [10] is investigated specification and verification of synchronous high-level digital systems using Dill (Digital Logic in LOTOS). What makes our approach different is the use of refinement calculus-based framework. The main advantage we gain is that each derivation step can be formally verified correct within the refinement calculus.

Overview of the Paper. In Section 2 we shortly revise the Action Systems formalism and its timed extension. In addition, we also revise the time constraint notation, and the refinement of timed action systems. In Section 3 we shortly describe the TACO protocol processor architecture. Then in Section 4 we introduce the formal model of the general TACO protocol processor and stepwisely derive the abstract model into a more concrete one. Finally, in Section 5 we end with some concluding remarks.

2 Action Systems

2.1 Actions

An *action* A is defined by

$$\begin{array}{lll}
A ::= & abort & (\textit{abortion, non-termination}) \\
 & \mid skip & (\textit{empty statement}) \\
 & \mid A_1 \;[]\; \ldots \;[]\; A_n & (\textit{non-deterministic choice}) \\
 & \mid A_1; \ldots ;A_n & (\textit{sequential composition}) \\
 & \mid x := e & (\textit{(multiple) assignment}) \\
 & \mid p \rightarrow A & (\textit{guarded action})
\end{array}$$

where A and A_i, $j = 1..n$, are actions; x is a variable or a list of variables; e is an expression or a list of expressions; p is a predicate (boolean condition).

The actions are defined using weakest precondition predicate transformers [3]. For example:

$$\mathrm{wp}(skip, Q) = Q \;\; , \mathrm{wp}(x := e, Q) = Q[e/x] \;\; .$$

Actions and action composition are considered atomic, which means that only their pre- and post-states are observable, and when they are chosen for execution they cannot be interrupted by external counterparts.

Variables. The variables which are assigned within the action A are called the *write variables* of A, denoted wA. The other variables present in the action A are called the *read variables* of A, denoted rA. The write and read variables form together the *access set* vA of A: $vA \widehat{=} wA \cup rA$.

Quantified Composition. A quantified composition of actions is defined by $[\bullet\ 1 \le i \le n : A_i] \hat{=} A_1 \bullet \ldots \bullet A_n$, where the bullet $\bullet$ denotes any of the composition operators, and n is the number of actions ($n \in \mathbb{N}$).

2.2 Timed Actions

A timed action system $\mathcal{A}t$ is, for example, of form:

```
sys $\mathcal{A}t$( )
|[
  sdelay $dA_i$
  var $vA$
  actions $A_i[\![dA_i]\!]: aA_i$
  init $vA, ct := vA_0, 0$
   do $[\ [\!]\ 1 \le i \le n : A_i]$ od
]|
```

where aA is any of the defined atomic actions, $[\ [\!]\ 1 \le i \le n : A_i] \hat{=} [\ [\!]\ 1 \le i \le n : (A_{w,i}\ [\!]\ A_{r,i})]\ /\!/\ [\ [\!]\ 1 \le i \le n : A_{o,i}]\ /\!/\ Pt$. A_i is called *a timed action* and Pt is *a progress time action* that forwards the current time. The parts of a timed action are: $A_{o,i}$ is an *operation* action, $A_{w,i}$ is a *write* action, and $A_{r,i}$ is a *release* action. An activity of an action A_i is performed in $A_{o,i}$. The result of the activity is postponed by a specified delay dA_i after which it is written in $A_{w,i}$ to a write variable wA_i. $A_{r,i}$ is only used when an timed action is disabled during the delay, that is, it prevents a timed action being *deadlocked*. The time domain $\mathbb{T}$ is dense ($\mathbb{R}_+$), and continuous ($\forall t_1 \exists t_2.(t_1 > t_2)$).

The operation of an action system is started by initialisation in which the variables are set to predefined values. Then, in the iteration, actions are selected for execution based on the composition operators and the enabledness of the actions. This is continued until there are no enabled actions after which the action system is temporarily stopped until some other action system enables it again.

2.3 Temporal Properties of Action Compositions

To describe and confirm temporal properties of hardware systems, we define a function that is used during the development phases: a *time constraint*. Time constraints are used to confirm the tenability of the timing during development phases.

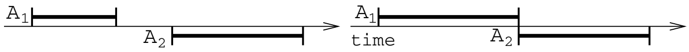
Definition 1. *The time constraint of timed actions is a function* $\square : \mathbb{T}^n \rightarrow \{T, F\}$ *, where* n *(*$\in \mathbb{N}^+$*) is the number of involved timed actions.*

Example 1. In this paper we use temporal relations to describe time constraints. Temporal relations describe the relative ordering of concurrent components, and they has been used to represent the behaviour of systems in time domain [11, 12, 13]. The temporal relations used in this study are shown in Fig. 1, and the time constraints based on the temporal relations are:

$$\square(A_1\ precedes\ A_2) \hat{=} ftA_1 < stA_2 \ ,$$
$$\square(A_1\ meets\ A_2) \hat{=} ftA_1 = stA_2 \ ,$$

where stA_i and ftA_i are the start and finish times of a timed action defined in Sect. 2.2. *End of example.*

Definition 2. *The time constraint of a timed action system* $\mathcal{A}$ *is a conjunction of action level time constraints:* $\square(\mathcal{A}) \hat{=} \bigwedge_{i=1}^{m} \square_i$ *where* $m \in \mathbb{N}^+$.



(a) A_1 *precedes* A_2 (b) A_1 *meets* A_2

Fig. 1. Temporal relations in a graphical form

2.4 Refinement

Action systems are meant to be developed in a stepwise manner within the *refinement calculus* framework [14]. The (atomic) action A is said to be *(correctly) refined* by action C, denoted $A \leq C$, if $\forall Q.(\mathrm{wp}(A,Q) \Rightarrow \mathrm{wp}(C,Q))$ holds.

Data Refinement. An action A on the variables a and u is *data-refined* by an action C on the variables c and u, denoted $A \leq_R C$, using an abstraction invariant $R(a,c,u)$, which is a boolean relation between the abstract variables a and the concrete variables c, if $\forall Q.(R \wedge \mathrm{wp}(A,Q) \Rightarrow \mathrm{wp}(C, \exists a.R \wedge Q))$ holds. The predicate $\exists a.R \wedge Q$ is a boolean condition on the program variables a and c.

Trace Refinement of Timed Action Systems. Consider timed action system:

```
sys $\mathcal{A}$( $g$ )
|[
 sdelay $dA$
 tconst $\Box(\mathcal{A})$
 var $a$
 actions $A[\![dA]\!]$: $aA$
 init $g,a := g0,a0$
  do $A$  od
]|
```

```
sys $\mathcal{C}$( $g$ )
|[
 sdelay $dC,dX$
 tconst $\Box(\mathcal{C})$
 var $c$
 actions $C[\![dC]\!]$: $aC$, $X[\![dX]\!]$ : $aX$
 init $g,c := g0,c0$
  do $C$ [] $X$  od
]|
```

where aA, aC, and aX are any of the atomic actions defined previously.

The system $\mathcal{C}$ is obtained from $\mathcal{A}$ by replasing the local variables a with new local variables c, the actions A_i with C_i, and adding a set of auxiliary action X_i into the system. The global variables g are not changed. The trace refinement theorem is used to prove refinement $\mathcal{A} \sqsubseteq \mathcal{C}$ [15] or its timed extension [16].

3 TACO Processor Architecture

The TACO protocol processor architecture [7] is based on the transport triggered architecture (TTA) [17, 18]. In TTA processor data transports are programmed and they trigger operations - traditionally operations are programmed and they trigger transports. A TTA processor is formed of functional units (FUs) that communicate via an interconnection network of data buses, controlled by an interconnection network controller unit. The FU's connect to the buses through modules called sockets.

A functional unit has input and output registers. FU operations are executed every time data is moved to a specific kind of FU input register, the trigger register. Each FU has one such register. Each functional unit in a TACO processor performs a specific protocol processing task. The FU operations can be, depending on the application to be implemented, bitstring matching, counting, timing comparisons, and random number generation. Figure 2 shows the functional view of one possible TACO protocol processor architecture. There can be more than one of each kind of FU's in a TACO processor.

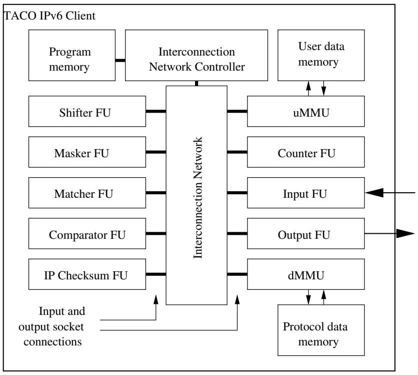


Fig. 2. The TACO IPv6client

4 Formal Model of the TACO Protocol Processor

The Timed Action System model of a general TACO protocol processor is given below. Our starting point is a conventional sequential program describing its behaviour. Through several refinement steps we end up a non-trivial model where functional units operate in parallel. For the sake of simplicity, we do not consider the external interface in this study.

The $\mathcal{T}acoPP$ is of form:

```
sys $\mathcal{T}acoPP(\ \dots\ )$
 |[
  const $laddr_i, addrTr_i, addrOp_i, addR_i$: $nat$
  var $abus.src, abus.dst, dbus, pword, tr_i, op_i, r_i, opcode_i$: $nat$; $rw, p$: $bool$
  actions $PP[\![dPP]\!]$: $Inc; [\ [\!]\ i \in \{1..n\} : Fu_{w,i}]; [\ [\!]\ i \in \{1..n\} : Fu_{r,i}]$
  init $abus.src, abus.dst, dbus, pword, tr_i, op_i, r_i, opcode_i := 0$; $rw, p := F, T$
   do $true \rightarrow PP$ od
  ]|
```

where *Inc* is:

$$Inc \widehat{=} pword := Pmem(pc); abus.src, abus.dst := pword[src], pword[dst]; pc := pc + 1 \ ,$$

and $Fu_{w,i}$ and $Fu_{r,i}$ are, in general:

$$\begin{aligned} &Fu_{w,i} \widehat{=} abus.src = addrR_i \rightarrow dbus := r_i \ , \\ &Fu_{r,i} \widehat{=} addrTr_i \leq abus.dst \leq (addrTr_i + laddr_i) \\ &\qquad \rightarrow tr_i, opcode_i := dbus, abus.dst - addrTr_i; r_i := Func(tr, op, opcode) \\ &\qquad [\!]\ abus.dst = addrOp_i \rightarrow op_i := dbus \ , \end{aligned}$$

where we have two input addresses and one output address: $addrTr_i$, $addrOp_i$, and $addrR_i$, respectively. The number of the addresses may vary between FUs.

The operation of the $\mathcal{T}acoPP$ protocol processor is as follows: A program word *pword* is read from the program memory *Pmem*, and then split on the source (*abus.src*) and destination (*abus.dst*) address lines. Then *abus.src* is compared with the FU's source addresses in $Fu_{w,i}$, and one of the FUs write data on the bus, *dbus*. Next, in $Fu_{r,i}$ the destination address is compared with the FUs' destination addresses. Depending on whether *abus.dst* equals a trigger ($addrTr_i$) or an operand ($addrOp_i$) addresses

```
sys $\mathcal{T}acoPP^1$( ... )
|[
 tconst $\Box_1(Inc$ meets $Rout_l$ meets $Fu_l)$
 const $laddr_l, addrTr_l, addrOp_l, addR_l$: $nat$
 var $abus.src, abus.dst, dbus, pword, tr_l, op_l, r_l, opcode_l$: $nat$; $p, w, a_l$: $bool$
 actions $Inc[\![dInc]\!]$: $p \to pword, abus.src, abus.dst := Pmem(pc), pword[src], pword[dst]$
                    $;pc, p, w := pc + 1, F, T$
         $Fu_l[\![dFu_l]\!]$: $a_l \wedge addrTr_l \le abus.dst \le (addrTr_l + laddr_l)$
                    $\to tr_l, opcode_l, a_l, w, p := dbus, abus.dst - addrTr_l, F, T, T$
                    $;r_l := Func(tr_l, op_l, opcode_l)$
                    $[\,]\ a_l \wedge abus.dst = addrOp_l \to op_l, a_l, p := dbus, F, T$
      $Rout_l[\![dRout_l]\!]$: $w \wedge abus.src = addrR_l \to dbus, w, a_l := r_l, F, T$
 init $abus.src, abus.dst, dbus, pword, tr_l, op_l, r_l, opcode_l := 0$; $a_l, p := F$
 do $Inc$ [] $Fu_l$ [] $Rout_l$ od
]|
```

Fig. 3. $\mathcal{T}acoPP^1$

a different activity is performed. The operation of the FU is only performed when there is a match with one of the trigger addresses.

Next we are going to developed the abstract TACO protocol processor model, in a stepwise manner, into a low-level representation. The goal of the refinement is to have a model in which the FUs are independent. This requires us to separate the operation of the FUs from the address decoding. Also the timing information is refined during the development giving us a more realistic view of the time that different operational blocks consume in their operation.

4.1 Formal Model of a Functional Unit

Let us start the development by decomposing the timed action PP such that $Fu_{w,i}$ and $Fu_{r,i}$ are located into their own timed actions. The refinement is straightforward as we retain the execution order by introducing three new boolean variables p, w, and a_l. The new timed actions are obtained by performing the following refinements:

$$PP \le Inc \;\; , skip \le Fu_i \;\; (i \in \{1..n\}) \;\; ,\text{and} \;\; skip \le Rout_i \;\; (i \in \{1..n\}) \;\; ,$$

where n is the number of the FUs. For the clarity of the presentation we concentrate only one of the FUs, FU_l ($l \in \{1..n\}$). The development of the other FUs follows the same procedure.

In the refinement we introduce a time constraint $\Box_1$ that reflect the original execution order in PP. It defines that the execution of Inc is followed immediately by the execution of $Rout_l$, and furthermore the execution of Fu_l starts right after the execution of $Rout_l$ is finished. The correctness of the performed trace refinement $TacoPP \sqsubseteq TacoPP^1$ (Fig. 3) and the following refinements can be proven by showing that all the conditions of the trace refinement of timed action systems are fulfilled. However, due to space limitation we do not give the detailed proofs of the refinements.

To further develop the FU towards a more concrete one, we separate the data read in phases from the computation, that is, we separate the address decoding and reading the data bus onto the FU's local variable from the operation of the FU. The execution order of the timed actions is not affected in the refinement, because Tr_l or Op_l cannot be enabled at the same time. Thus, they are mutually exclusive. Furthermore, a new

```
sys $\mathcal{T}acoPP^{2}(\ \ldots\ )$
|[
 tconst $\Box_1(Inc$ meets $Rout_l$ meets $(Tr_l \vee Op_l)), \Box_2(Tr_l$ meets $Fu'_l)$
 const $laddr_l, addrTr_l, addrOp_l, addR_l$: $nat$
 var $abus.src, abus.dst, dbus, pword, tr_l, op_l, r_l, opcode_l$: $nat$; $p, w, a_l, b_l$: $bool$;
 actions $Inc[\![dInc]\!]$: $p \rightarrow pword, abus.src, abus.dst := Pmem(pc), pword[src], pword[dst]$
                         $; pc, p, w := pc + 1, F, T$
           $Tr_l[\![dTr_l]\!]$: $a_l \wedge addrTr_l \leq abus.dst \leq (addrTr_l + laddr_l)$
                         $\rightarrow tr_l, opcode_l, a_l, b_l := dbus, abus.dst - addrTr_l, F, T$
          $Op_l[\![dOp_l]\!]$: $a_l \wedge abus.dst = addrOp_l \rightarrow op_l, a_l, p := dbus, F, T$
          $Fu'_l[\![dFu'_l]\!]$: $b_l \rightarrow r_l := Func(tr_l, op_l, opcode_l); b_l, p := F, T$
       $Rout_l[\![dRout_l]\!]$: $w \wedge abus.src = addrR_l \rightarrow dbus, w, a_l := r_l, F, T$
 init $abus.src, abus.dst, dbus, pword, tr_l, op_l, r_l, opcode_l := 0$; $p, w, a_l, b_l := F$
  do $Inc\ [\!]\ Tr_l\ [\!]\ Op_l\ [\!]\ Fu'_l\ [\!]\ Rout_l$ od
]|
```

Fig. 4. $\mathcal{T}acoPP^2$

boolean variable b_l is introduced to sequence the execution of Tr_l and Fu'_l. The former reads data in and the latter performs the operation of the FU. The refinements are:

$$Fu_l \leq Fu'_l \ , \ skip \leq Tr_l \ , \ \text{and } skip \leq Op_l \ .$$

Because Tr_l and Op_l are mutually exclusive, and Tr_l enables Fu_l the execution order of the timed actions is not affected. This is also reflected by the introduced time constraint $\Box_2$. We have a trace refinement $\mathcal{T}acoPP^1 \sqsubseteq \mathcal{T}acoPP^2$ (Fig. 4).

Next we separate the address decoding performed in the guards of the timed actions Tr_l and Op_l into their own timed actions. We also introduce a buffering assignment between the operation of the FU and its result register. We have refinements:

$$Tr_l \leq Trin_l \ , \ skip \leq Tr'_l \ , \ skip \leq Trload_l \ , \ skip \leq R_l \ ,$$
$$Op_l \leq Opin_l \ , \ skip \leq Opload_l \ , \ skip \leq Op'_l \ , \ \text{and } Fu'_l \leq Fu''_l \ ,$$

where we introduced new intermediate variables $trload_l$ and $opload_l$ that buffer the address match signals, and new boolean variables c_l, d_l, e_l that retain the correct execution order. The time constraints are also refined to meet the execution order of the new timed actions. Furthermore, we decomposed the time constraint $\Box_2$ into two parts. We have a trace refinement $\mathcal{T}acoPP^2 \sqsubseteq \mathcal{T}acoPP^3$ (Fig. 5).

The next refinement requires a little more attention than the previous ones. In this refinement we introduce a new boolean variable *clk* that sequence the operation of the timed actions, that is, the operation of the action is divided into *read* and *write* phases.

The earlier refinements grounded this refinement step by decomposing the operational parts of the FU from the data read in parts. Thus, giving us a possibility to strengthen only the guards of the data read in actions, that is, to strengthen the guards of those actions which store the data used in the computation. With operational parts we mean those timed actions that perform some sort of computation: address decoding and the functionality of the FU. We have refinements:

$$Inc \leq Inc', \ Tr'_l \leq Tr''_l, \ Trload_l \leq Trload'_l, \ R_l \leq R'_l,$$
$$Opload_l \leq Opload'_l, \ Op'_l \leq Op''_l, \ \text{and } skip \leq Clk \ .$$

```
sys $\mathcal{T}acoPP^3(\ \ldots\ )$
|[
 tconst $\Box_1(Inc\ \textbf{meets}\ Rout_l\ \textbf{meets}\ (Trin_l \vee Opin_l))$
        $\Box_{2a}(Trin_l\ \textbf{meets}\ Trload_l\ \textbf{meets}\ Tr'_l), \Box_{2b}(Tr'_l\ \textbf{meets}\ Fu''_l\ \textbf{meets}\ R_l)$
        $\Box_3(Opin_l\ \textbf{meets}\ Opload'_l\ \textbf{meets}\ Op'_l)$
 const $laddr_l, addrTr_l, addrOp_l, addR_l\colon nat$
 var $abus.src, abus.dst, dbus, pword, tr_l, op_l, r_l, r'_l, opcode_l, trload_l, opload_l\colon nat$
     $p, w, a_l, b_l, c_l, d_l, e_l, trload, opload\colon bool$
 actions     $Inc[\![dInc]\!]\colon$ $p \rightarrow pword := Pmem(pc); abus.src, abus.dst := pword[src], pword[dst]$
                             $; pc, p, w := pc + 1, F, T$
          $Trin_l[\![dTrin_l]\!]\colon$ $a_l \wedge addrTr_l \leq abus.dst \leq (addrTr_l + laddr_l)$
                             $\rightarrow myaddrTr_l, trdbus_l, operation_l := newline, dbus, abus.dst - addrTr_l$
                             $; a_l, d_l := F, T$
                             $[\!]\ a_l \wedge \neg(addrTr_l \leq abus.dst \leq (addrTr_l + laddr_l))$
                             $\rightarrow myaddrTr_l, a_l, d_l := F, F, T$
         $Trload_l[\![dTrl_l]\!]\colon$ $d_l \rightarrow trload_l, d_l := myaddrTr_l, F$
             $Tr'_l[\![dTr'_l]\!]\colon$ $trload_l \rightarrow opcode_l, tr_l, b_l, p := operation_l, trdbus_l, T, T$
          $Opin_l[\![dOpin_l]\!]\colon$ $a_l \wedge addrOp_l = abus.dst \rightarrow myaddrOp_l, opdbus_l, a_l, e_l := T, dbus, F, T$
                             $[\!]\ a_l \wedge addrOp_l \neq abus.dst \rightarrow myaddrOp_l, a_l := F, F$
        $Opload'_l[\![dOpl_l]\!]\colon$ $e_l \rightarrow opload_l, e_l := myaddrOp_l, F$
             $Op'_l[\![dOp'_l]\!]\colon$ $opload_l \rightarrow op_l, p := opdbus_l, T$
             $Fu''_l[\![dFu''_l]\!]\colon$ $b_l \rightarrow r'_l := Func(tr_l, op_l, opcode_l); b_l, c_l := F, T$
               $R_l[\![dR_l]\!]\colon$ $c_l \rightarrow r_l, c_l := r'_l, F$
          $Rout_l[\![dRout_l]\!]\colon$ $w \wedge abus.src = addrR_l \rightarrow dbus, w, a_l := r_l, F, T$
 init $abus.src, abus.dst, dbus, pword, tr_l, op_l, r_l, r'_l opcode_l, trload_l, opload_l := 0$
      $p, w, a_l, b_l, c_l, d_l, e_l := F$
 do $Inc\ [\!]\ Trin_l\ [\!]\ Trload_l\ [\!]\ Tr'_l\ [\!]\ Opin_l\ [\!]\ Opload_l\ [\!]\ 0p'_l\ [\!]\ Fu''_l\ [\!]\ R_l\ [\!]\ Rout_l$ od
]|
```

Fig. 5. $\mathcal{T}acoPP^3$

The correctness of a trace refinement $\mathcal{T}acoPP^3 \sqsubseteq \mathcal{T}acoPP^4$ (Fig. 6) is shown by choosing an abstract relation R_a:

$$\begin{aligned}R_a \widehat{=} &(p = clk \wedge p') \wedge (\neg p = \neg p') \wedge (trload = clk \wedge trload') \wedge (\neg trload = \neg trload')\\ &\wedge (opload = clk \wedge opload') \wedge (\neg opload = \neg opload') \wedge (d_l = \neg clk \wedge d'_l) \wedge (\neg d_l = \neg d'_l)\\ &\wedge (e_l = \neg clk \wedge e'_l) \wedge (\neg e_l = \neg e'_l) \wedge (c_l = clk \wedge c'_l) \wedge (\neg c_l = \neg c'_l)\ .\end{aligned}$$

As we perform a refinement that sequence the operation of the timed action system, the timed actions whose operation is constrained by the time constraint **meets** no longer holds. Therefore, we need to refine **meets** to **precedes**, so that the new operation sequence is reflected. The time constraint **precedes** allows a time gap between the execution of timed actions. Now, we have reached a synchronous representation in which the functional units are independent, and thus considered parallel. The independence of the FUs is ensured by the input and output sockets whose operation is sequenced by the introduced boolean variables. Although, FUs are independent and thus a parallel operation possible, the number of buses does not yet support parallel operation of functional units. Therefore, to increase the number of buses we need to perform a data refinement in which all the involved variables are replaced with a vector of the same type. We have, as an example for the variables *pword*, *dbus*, *abus.src*, and *abus.dst* an abstract relation R_b for three bus TACO protocol processor:

$$R_b \widehat{=} pword = 3pword \wedge dbus = dbus[3] \wedge abus.src = abus.src[3] \wedge abus.dst = abus.dst[3]\ ,$$

where $pword3$ contains three source and destination addresses, and $dbus[3]$, $abus.src[3]$, and $abus.dst[3]$ are vectors containing one item per bus.

$$
\begin{array}{l}
\mathbf{sys}\ \mathcal{T}acoPP^4(\ \ldots\ )\\
|[\\
\ \mathbf{tconst}\ \square_1(Inc'\ \mathbf{meets}\ Rout_l\ \mathbf{meets}\ (Trin_l \vee Opin_l)), \square_{2a}(Trin_l\ \mathbf{precedes}\ Trload'_l\ \mathbf{precedes}\ Tr''_l)\\
\qquad\qquad \square_{2b}(Tr''_l\ \mathbf{meets}\ Fu''_l\ \mathbf{precedes}\ R'_l), \square_3(Opin_l\ \mathbf{precedes}\ Opload'_l\ \mathbf{precedes}\ Op'_l)\\
\ \mathbf{const}\ laddr_l, addrTr_l, addrOp_l, addR_l\colon nat\\
\ \mathbf{var}\ abus.src, abus.dst, dbus, pword, tr_l, op_l, r_l, r'_l opcode_l\colon nat\\
\qquad clk, p', w, a_l, b_l, c'_l, d'_l, e'_l, trload', opload', iclk, aclk_l, bclk_l, cclk_l, dclk_l, eclk_l\colon bool\\
\ \mathbf{actions}\ \begin{array}[t]{rl}
Clk[\![dClk]\!]\colon & \neg clk \to clk := T\ [\!]\ clk \to clk := F\\
Inc'[\![dInc]\!]\colon & p' \wedge clk \wedge iclk \to pword := Pmem(pc)\\
 & ; abus.src, abus.dst, pc, p', w, iclk := pword[src], pword[dst], pc+1, F, T, F\\
 & [\!]\ \neg clk \wedge \neg iclk \to iclk := T\\
Trin_l[\![dTrin_l]\!]\colon & a_l \wedge addrTr_l \le abus.dst \le (addrTr_l + laddr_l)\\
 & \to myaddrTr_l, trdbus_l, a_l, d'_l := T, dbus, F, T\\
 & ; operation_l := abus.dst - addrTr_l\\
 & [\!]\ a_l \wedge \neg(addrTr_l \le abus.dst \le (addrTr_l + laddr_l))\\
 & \to myaddrTr_l, a_l, d'_l := F, F, T\\
Trload'_l[\![dTrl_l]\!]\colon & d'_l \wedge \neg clk \wedge aclk_l \to trload'_l, d'_l, aclk_l := myaddrTr_l, F, F\\
 & [\!]\ clk \wedge \neg aclk_l \to aclk_l := T\\
Tr''_l[\![dTr''_l]\!]\colon & trload'_l \wedge clk \wedge bclk_l \to opcode_l, tr_l, b_l, bclk_l, p' := operation_l, trdbus_l, F, T, T\\
 & [\!]\ \neg clk \wedge \neg bclk_l \to bclk_l := T\\
Opin_l[\![dOpin_l]\!]\colon & a_l \wedge addrOp_l = abus.dst \to myaddrOp_l, opdbus_l, a_l, e'_l := T, dbus, F, T\\
 & [\!]\ a_l \wedge addrOp_l \neq abus.dst \to myaddrOp_l, a_l := F, F\\
Opload'_l[\![dOpl_l]\!]\colon & e'_l \wedge \neg clk \wedge cclk_l \to opload'_l, e'_l, cclk_l := myaddrOp_l, F, F\\
 & [\!]\ clk \wedge \neg cclk_l \to cclk_l := F\\
Op''_l[\![dOp''_l]\!]\colon & opload'_l \wedge clk \wedge dclk_l \to op_l, p', dclk_l := opdbus_l, T, F\\
 & [\!]\ \neg clk \wedge \neg dclk_l \to dclk_l := T\\
Fu''_l[\![dFu''_l]\!]\colon & b_l \to r'_l := Func(tr_l, op_l, opcode_l); b_l, c'_l := F, T\\
R'_l[\![dR_l]\!]\colon & c'_l \wedge clk \wedge eclk_l \to r_l, c'_l, eclk_l := r'_l, F, F\\
 & [\!]\ \neg clk \wedge \neg eclk_l \to eclk_l := T\\
Rout_l[\![dRout_l]\!]\colon & w \wedge abus.src = addrR_l \to dbus, w, a_l := r_l, F, T
\end{array}\\
\ \mathbf{init}\ abus.src, abus.dst, dbus, pword, tr_l, op_l, r_l, r'_l, opcode_l := 0\\
\qquad clk, p', w, a_l, b_l, c'_l, d'_l, e'_l, myaddrTr_l, myaddrOp_l, iclk, aclk_l, bclk_l, cclk_l, dclk_l, eclk_l := F\\
\ \mathbf{do}\ Clk\ [\!]\ Inc'\ [\!]\ Trin_l\ [\!]\ Trload'_l\ [\!]\ Tr''_l\ [\!]\ Opin_l\ [\!]\ Opload'_l\ [\!]\ 0p''_l\ [\!]\ Fu''_l\ [\!]\ R'_l\ [\!]\ Rout_l\ \ \mathbf{od}\\
]|
\end{array}
$$

Fig. 6. $\mathcal{T}acoPP^4$

5 Conclusions

We presented in this study a formal specification of a general TACO protocol processor model. The used formal method was Timed Action System, which allows us, in addition to functional properties, to model also the temporal properties of the system. The derived general TACO protocol processor model completes the existing TACO SystemC, VHDL, and Matlab models by offering tools to formally verify both the functional and temporal properties of TACO protocol processor architectures. The SystemC model is used for system-level simulations, the VHDL model for synthesising architectures, and the Matlab model for estimating physical parameters at the system level.

The advantage of using Timed Action Systems throughout the development of the TACO protocol processor is that we can formally verify each derivation step correct within the refinement calculus. During the derivation steps the time constrains were created and refined to reflect the temporal properties of the refined timed actions.

The future work studies include, for example, developing a formal model of the whole TACO protocol processor, creating a library that contains formal models of the existing FU models, and importing timing information, obtained from the synthesised VHDL models, into the formal framework. The timing information can be used to verify the temporal properties of the TACO protocol processor.

References

1. Back, R.J., Sere, K.: From Modular Systems to Action Systems. In: Proc. of Formal Methods Europe '94, Spain, Lecture notes in computer science, Springer-Verlag (1994)
2. Back, R.J., Kurki-Suonio, R.: Decentralization of Process Nets with Centralized Control. In: Procs. of the 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing. (1983) 131–142
3. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall International (1976)
4. Seceleanu, T.: Systematic Design of Synchronous Digital Circuits. PhD thesis, Turku Centre for Computer Science (2001)
5. Plosila, J.: Self-Timed Circuit Design - The Action System Approach. PhD thesis, University of Turku (1999)
6. Westerlund, T., Plosila, J.: Formal Timing Model for Hardware Components. In: Proceedings of the 22nd NORCHIP Conference, Norway (2004) 293–296
7. Virtanen, S.: A Framework for Rapid Design and Evaluation of Protocol Processors. PhD thesis, University of Turku (2004)
8. Back, R.J., Martin, A.J., Sere, K.: Specifying the Caltech Asynchronous Microprocessor. Sci. Comput. Program. **26** (1996) 79–97
9. Plosila, J., Sere, K.: Action Systems in Pipelined Processor Design. In: Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems. (1997) 156–166
10. He, J., Turner, K.J.: Specification and verification of synchronous hardware using LOTOS. In: Proc. Formal Methods for Protocol Engineering and Distributed Systems. (1999) 295–312
11. Kim, H., Beerel, P.A.: Relative timing based verification of timed circuits and systems. In: Proc. International Workshop on Logic Synthesis. (1999)
12. Stevens, K., Ginosar, R., Rotem, S.: Relative Timing. In: Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems. (1999) 208–218
13. Rotem, S., Stevens, K., Ginosar, R., Beerel, P., Myers, C., Yun, K., Kol, R., Dike, C., Roncken, M., Agapiev, B.: RAPPID: An Asynchronous Instruction Length Decoder. In: Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems. (1999) 60–70
14. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer-Verlag (1998)
15. Back, R.J., von Wright, J.: Trace Refinement of Action Systems. In: International Conference on Concurrency Theory. (1994) 367–384
16. Westerlund, T., Plosila, J.: Towards a Timed Refinement Calculus for Hardware Systems. Technical Report TR669, Turku Centre for Computer Science (2004)
17. Corporaal, H.: Microprocessor Architectures - from VLIW to TTA. John Wiley and Sons Ltd. (1998)
18. Tabak, D., Lipovski, G.J.: MOVE Architecture in Digital Controllers. IEEE Transactions on Computers **29** (1980) 180–190

Tuning a Protocol Processor Architecture Towards DSP Operations

Jani Paakkulainen[1], Seppo Virtanen[1,2], and Jouni Isoaho[1]

[1] Department of Information Technology, University of Turku,
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
jani.paakkulainen@utu.fi

[2] Embedded Systems lab, Turku Centre for Computer Science,
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland

Abstract. In this paper we present an experiment in enhancing our transport triggered protocol processor hardware platform to support DSP applications. Our focus is on integrating support for both application domains into a single processor without loss of performance in either domain. Such a processor could be taken advantage of in applications like Voice-over-IP communication using hand-held devices, where functionality is needed from both domains. As our first step in bridging the gap between the protocol processing and DSP domains we implement support for FIR filtering. We analyze four different architectural instances for implementing FIR filters according to their performance and bus utilisation. We were able to determine that protocol processing and DSP operations can be executed in parallel very efficiently. The implementations were verified with VHDL simulations and synthesis using 0.18 μm CMOS technology.

1 Introduction

Increasing system complexity and performance requirements are forcing system designers to constantly find better ways of integrating more functionality into their devices while keeping up with stringent requirements for power consumption and chip area. The emergence of System-on-Chip and Network-on-Chip has made it possible to create very complex designs with specific parts of the system (e.g. a DSP or a network interface subsystem) designed by a third party. Such an approach relieves local design teams of some of the design work at the cost of potentially more effort needed in tasks like power optimization and on-chip communication design. For example, in designing a special-purpose multimedia chip, a quick solution in terms of functional design would be to import the networking and DSP functionality into the architecture as intellectual property (IP) blocks designed by third parties. In this kind of a design it is likely that the IP blocks will provide a superset of the functionality required for the multimedia chip. This would also produce an unnecessary area and power consumption overhead for the overall design process. If however an IP block with only the bare minimum required networking and DSP capabilities could be obtained, clearly there would be foreseeable advantages to be gained in terms of area use, power consumption and also data throughput (no on-chip data transportation between the separate networking and DSP modules would be required). This is especially true for applications like

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 132–141, 2005.

Voice-Over-Internet-Protocol (VoIP) communication using hand-held devices: support and optimization for special functionality from both the DSP and the protocol processing application domains is needed while the designer faces stringent constraints in terms of device size and power consumption. In previous work we have developed a TTA (transport triggered architecture; see [1, 2]) based hardware platform for designing application optimized programmable protocol processor architectures, and a framework for designing processors adhering to the hardware platform specification [3].

In this paper we start exploring possibilities for augmenting our hardware platform to support DSP operations in addition to the pre-existing support for protocol processing. Our long-term goal is the ability to design a modular, programmable and application-optimized core IP block for any application requiring both protocol processing and DSP functionality (like VoIP). As our first step towards bridging the gap between the protocol processing and DSP domains, in this paper we enhance the TACO (tools for application-specific hardware/software codesign) hardware platform to support finite impulse response (FIR) filtering. For this, it is necessary to implement support for the *multiply-and-accumulate (MAC)* operation. Initially we take an approach that requires the least modifications to our hardware platform. Thereafter we start tuning the existing modules of the TACO architecture to better cope with the co-existence of the protocol processing and DSP application domains in a single processor. We conclude the paper with experimental results demonstrating that protocol processing and DSP operations can be executed in parallel very efficiently using our enhanced hardware platform. We verify the additions and modifications to the hardware platform with VHDL simulations and synthesis using 0.18 μm CMOS technology.

1.1 Related Work

Methodologies for ASIP (application-specific instruction-set processor) design are becoming increasingly popular in embedded processor design, especially for DSP applications. A clear sign of this trend is the appearance of commercial ASIP design tool suites like LISATek (CoWare) [4] and Chess/Checkers (Target Compiler Technologies) [5] on the market scene. Also in the academic community the research concerning ASIPs is quite active. Here we briefly outline the most relevant approaches in terms of the work presented in this paper.

In [6] a common machine description is used for both a compiler and a core generator. The core generator generates simulation and synthesis models for an architectural template called STA (synchronous transfer architecture). According to [6], STA is a simplification of TTA optimized for the predictable execution environment of DSP. The *MOVE* framework [1, 7] consists of a set of tools for automatic design space exploration and synthesis of hardware and software. The tools operate on a parametric TTA template that consists of funtional units (FUs) that implement combinations of general purpose operations. A designer can also define user modules called special function units (SFUs) for use in the template.

The *MOVE* framework has been applied to DSP ASIP design in e.g. [8]. Another approach in applying the TTA paradigm to DSP ASIP design is presented in [9]. However, we are not aware of any approaches besides our own in applying the TTA paradigm to protocol processing; hence, we are also not aware of other TTA approaches that incorporate hardware support for both the DSP and the protocol processing application domains. In comparison to the mentioned TTA DSP approaches, also the TACO protocol processor design framework relies on a TTA based hardware platform in designing application-specific processors. However, in the TACO framework the sequences of operations chosen for hardware implementation are considerably larger than in traditional ASIP; in TACO the emphasis is not on detecting recurring general purpose code sequences but in identifying frequently needed domain-specific functions in the target application. Also, TACO processors are not general purpose processors enhanced with special execution units for the detected functionality, but are constructed solely of special execution units and no general purpose processing elements. A further discussion on related ASIP methodologies and a comparison to the TACO framework can be found in [3].

2 The TACO Framework

The starting point in the TACO design flow is a high level application description or specification. The development of the application software guides the processor design work so that the processing requirements of the application determine the hardware architecture of a protocol processor [10, 11]. The approach is quite different from what is found in most commercial protocol processors available today. They are most often multiprocessors with high performance general purpose computing elements.

The TACO protocol processor architecture, as seen in Fig. 1, is based on transport triggered architectures (TTA) [1, 2]. The most important difference between TTAs and traditional processors is that in TTAs operations occur as side effects of programmed data transports. In traditional processors the operations are programmed, and data transports occur as side effects of the programmed operations. A TTA based processor is composed of functional units (FUs) that communicate via an interconnection network. This network of data buses is controlled by a special-purpose controller unit. The FUs are connected to the buses through modules called sockets. It was observed in the TACO project that this kind of modularity facilitates both component reuse and hardware design automation.

Although TTA based, the TACO architecture differs from existing TTA approaches in many ways. The reader is referred to [3] for a detailed discussion on some of the differences. Within the scope of this paper we limit ourselves to stating one important characteristic of the TACO architcture: all TACO functional units are Special Functional Units (SFUs, as seen in Fig. 1). There are no general purpose FUs as described in e.g. [1] in the TACO architecture. SFUs are FUs with a distinct application-domain specific task to execute, and usually they can not efficiently be used in some other application domain. The application domain for the TACO SFUs has thus far always been protocol processing.

The SFUs in TACO processors have a varying number of input and output registers with programmable addresses. SFU operations are executed every time data is moved to a specific kind of input register, the trigger register. Each SFU has one trigger register, and each SFU performs a specific protocol processing task or operation (for example,

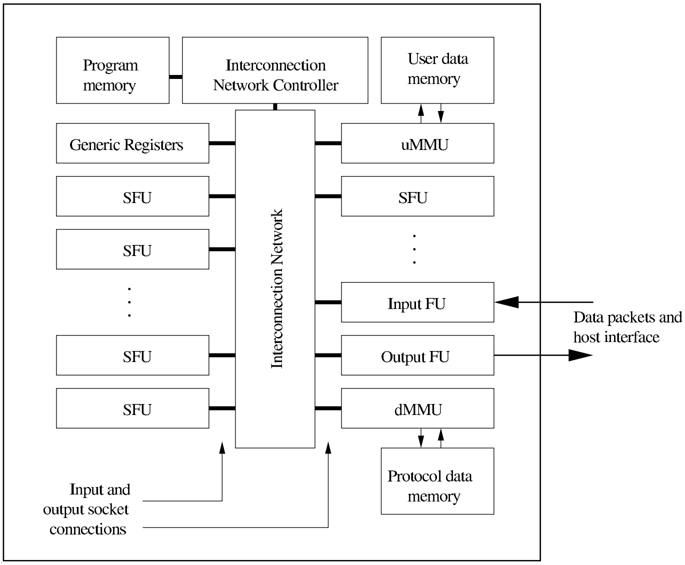


Fig. 1. A functional view of the TACO architecture

checksum calculation, bitstring matching and masking inside data words, and logical comparisons). The TACO architecture (see Fig. 1) is in our view a template for protocol processors, instantiated for a specific protocol, or family of protocols. Extensive component and module reuse is typical for TACO design projects.

3 From Protocol Processing to DSP: Tuning the Architecture

The benefit of TTA based platforms is their modularity and scalability. Functional units can be added to an architecture or they can be refined and changed as long as they provide the same interface to the sockets connecting them to the interconnection network. From this follows also module reusability, which we consider to be a very important strong-point of our TACO hardware platform and design methodology: all our existing functional units are similar in terms of interface type, and they have been designed with common guidelines. When designing new functional units for the hardware platform, most often the first phase of the design process is to look among our existing FUs to find the one whose implementation and provided functionality is closest to the requirements of the new unit to be designed. Such an existing functional unit serves as a "design template" for the design process of the new FU, making the process quite fast and straightforward.

3.1 Taking Advantage of Existing Modules in MAC FU Design

The multiply-and-accumulate (MAC) operation between two long sequences of data operands is traditionally regarded as one of the most important low-level operations needed in DSP calculations. For this reason the operation is most often supported by hardware in digital signal processors. The calculation itself is quite straightforward, and implementational differences arise mostly from the way the data operands are transported to the unit performing the calculation. The MAC operation is vital to e.g. digital filtering, in which one of the data sequences carries samples of a digitized signal, and the other one carries filter parameters called taps. For example, the *n*th output sample *y(n)* of a finite impulse response (FIR) filter is defined by the equation

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k), \tag{1}$$

where *x(n)* is the *n*th input sample, and *h(k)* is a sequence of *N* filter taps. *N* MAC operations and $2N$ data transports are needed for calculating one output sample of the filter, which clearly demonstrates the importance of an efficient MAC implementation to applications of the DSP domain.

We chose our existing Internet checksum FU (Fig. 2) to act as a starting point for the process of designing a MAC functional unit for our hardware platform. For a MAC functional unit implementation at least three distinct executional blocks are needed: a multiplier, an adder and an accumulator. The existing Internet checksum FU already had an arithmetic block and an accumulator. By keeping the accumulator and modifying the arithmetic block to calculate the required addition and multiplication operations the checksum FU almost changes into a MAC unit. To reach the MAC unit structure depicted in Fig. 2, some additional minor modifications are still needed: the combined accumulator and result register bank needs to be divided into two different registers, some control logic is attached between the output of the adder and the input of the result register, the output inverters are removed, and the main control of the FU is modified. Also the unnecessary input register (operand) is removed.

As in any fixed point system, also in this setup the designer has to decide beforehand how the two input data sequences are handled. We assume that both inputs are normalized into the range $[-1,1]$. When these inputs are decoded by the hardware into a fixed point representation, the leftmost bit is interpreted as a sign bit and the rest of the bits are interpreted as magnitude. The word size of the data in the input sequences can be specified to be narrower than the word size on the TACO interconnection network data buses, which makes it possible to execute e.g. the filtering of 16 bit input samples in parallel with a 32 bit checksum calculation. Naturally in such a setup the data sequences need to be formatted correctly to meet the MAC unit configuration: all data is still transported at the specified bus width in the processor, so the data sequences need to be padded with zeros from the most significant bit to the bit preceeding the sign bit. In terms of specifying processor architectures, it is up to the designer to decide the register widths used for the accumulator and the result register, just as the designer is currently already required to decide the system-wide word size of the processor (like the MAC unit's word size, also the TACO processor word size is parametrizable). At the time of fixing the MAC unit accumulator width it is also necessary to take into account the lengths of the data sequences (e.g. the number of filter taps) as well as their magnitudes and internal dynamics.

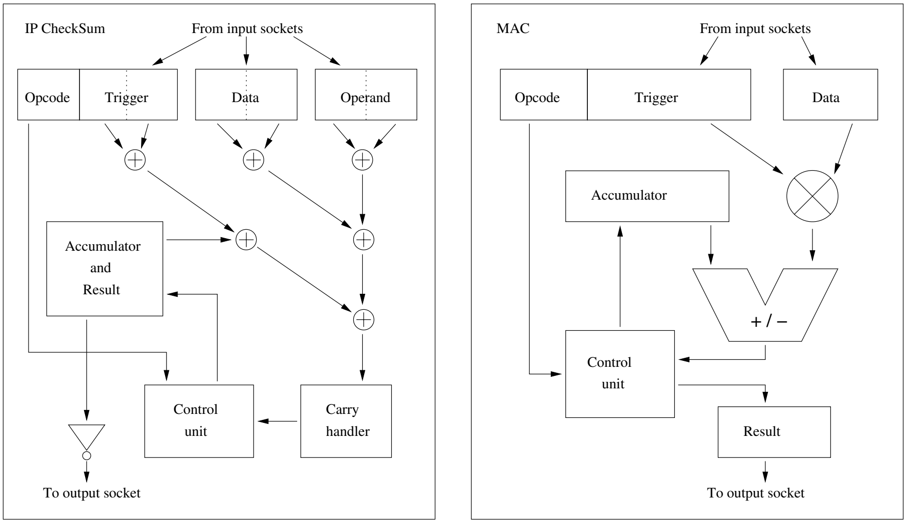


Fig. 2. An Internet Checksum FU and a Multiply-and-Accumulate FU

3.2 Data Activity Based Resource Allocation

As mentioned previously, a typical application of the MAC function in digital signal processing is filtering. In the following we explore four different FIR filtering implementations using our existing protocol processing functional units and the new MAC functional unit. This also involves exploring whether potential speed-ups could be achieved by adding new features like address counters into our existing memory management units. An important characteristic of the TACO hardware platform is that there is no limit for the number of memory managers and associated memory blocks that can be incorporated into an architecture. We exploit this characteristic in the following by using two separate memory managers and associated memory blocks for transporting data to the new MAC FU.

In our first TACO FIR implementation we specify an architecture in which we take full advantage of the data transportation capabilities of our interconnection buses. This means that we include as many interconnection buses and functional units into the architecture as required for being able to constantly transport all required operands to the input registers of the MAC FU. For this implementation (labeled Arc I in the results shown in Table 1), five different FUs and five interconnection network data buses are needed. Four of the required functional units, namely two Counter FUs, a uMMU(user memory management unit) and a dMMU(data memory management unit) already existed and had been tested in our earlier protocol processing implementations of the TACO architecture. With five buses, the MAC FU is able to perform one multiply-and-accumulate calculation in each clock cycle. After performing all the required calculations needed to produce one output sample, one clock cycle is consumed by moving the output sample out of the result register and for initializing the counter FUs and the MAC FU for the next round of calculations.

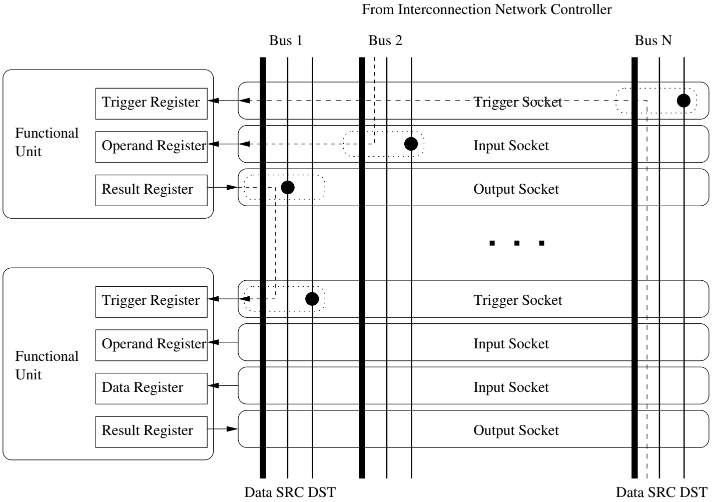


Fig. 3. Connecting functional units through buses and sockets

In the second implementation (Arc II in Table 1) there is one interconnection bus less than in the first one. This reduction in the bus count is achieved by making certain small modifications into the uMMU (user memory management unit): by adding an extra address counter into the uMMU one counter FU can be eliminated. In terms of implementing only FIR filtering this might seem to be an efficient design choice; however, if FIR filtering is implemented as part of a larger application there is a penalty to be paid for changing the uMMU. For example, if it were necessary to simultaneously perform some other tasks in the same architecture besides FIR filtering, for example processing the IPv6 protocol, the original uMMU implementation would still be needed, Thus, instead of a single uMMU in the architecture it would be necessary to include two slightly differing uMMU implementations.

Both of the implementations described above are able to use the interconnection buses very efficiently, but on the other hand this also means that no other tasks can be executed in parallel with the FIR filtering. Depending on which is determined to be more important in terms of the hardware architecture and its target operating environment, the designer could equally well select either one of these two implementations; one performs the filtering with fewer buses (Arc II), while the other one does it with fewer functional units (Arc I).

The third implementation (ARC III in Table 1) is a compromise between processing speed and the number of interconnection buses. The FUs incorporated into this implementation are the same as the ones used in Arc I. In Arc III only two buses are used by the filtering process; thus, the designer can either add more buses into the architecture and so make parallel task execution possible for the programmer. In comparison with e.g. Arc I, three more buses could be added into the architecture with similar hardware costs and with the option of performing the filtering in parallel with some protocol processing tasks like e.g. IPv6 header analysis.

Table 1. Architecture comparisons for N-tap FIR filter, data sample rate 44.1 kHz

Architecture instance	Cycle equation	Taps N	Cycles per sample	Required clk MHz	FU used	Bus used
Arc I	N+1	35	36	1.6	5	5
		256	257	11.4	5	5
Arc II	N+1	35	36	1.6	4	4
		256	257	11.4	4	4
Arc III	3×N+2	35	107	4.8	5	2
		256	770	34.0	5	2
Arc IV	5×N+4	35	179	7.9	5	1
		256	1284	56.7	5	1

The last implementation (Arc IV in Table 1) requires only one interconnection bus. Arc IV is actually a variation of Arc III; If the target architecture will anyway contain at least two buses, the designer is free to choose whether to reserve one (Arc IV) or two (Arc III) for the filtering task while the remaining buses can be used in parallel with the filtering for some other tasks. If the number of filter taps or the sample rate of the input data increases, Arc IV is naturally the first implementation to reach the limits of its capabilities in FIR filtering; in such a case, it may be necessary to reserve two buses for the filtering process.

3.3 Synthesis and Verification

The MAC functional unit and the rest of the modules were synthesized using 0.18 μm CMOS technology at a target clock speed of 250 MHz. Two different input word lengths were used to find out the dependence between area and input word length. With 16 bit inputs the area was approximately 32 900 μm^2 ($\sim$ 2 700 gates) and with 32 bit inputs the area was approximately 91 500 μm^2 ($\sim$ 7 500 gates). With 32 bit word size the cost of having a larger multiplier becomes quite obvious. For 16 bit inputs, the area is in the same order of magnitude as the average FU size of our existing units (20 000 μm^2 or 1 700 gates). With 32 bit inputs the MAC FU consumes three times more area than the one with 16 bit inputs, but still the size is within the limits of our typical large FUs. Our previously implemented TACO architectures for protocol prosessing tasks have consumed area in the range of 0.2 - 1.0 mm^2 (17 - 85 kgates), not including memories. The area increase is approximately 5-10 % when a MAC FU is inserted into an average TACO protocol processor architecture. This kind of area increase is quite acceptable when a new special functional unit is integrated into the TACO architecture.

The test material for VHDL simulations was generated using Matlab, and the simulation results were checked against Matlab simulation results. The simulation results for MAC-enhanced TACO architectures were also verified against results obtained on a Texas Instruments TMS320C5510 Digital Signal Processor. The VHDL simulation results were identical with the results from the signal prosessor, and both results were nearly identical when compared to results calculated using Matlab. In 10 000 input data samples just a couple of the results had differences in the least significant bit, and the difference traced back into truncation errors in fixed point systems.

4 Conclusions and Future Work

We presented an experiment in enhancing our TTA protocol processor architecture towards DSP applications, with focus on integrating support for both application domains into a single processor without loss of performance in either domain. We analyzed four different architectural instances for implementing FIR filters in terms of their performance and bus utilization, and concluded that application software can be written for our hardware platform in a way that allows efficient parallel execution of both protocol processing and DSP operations.

The process of designing and implementing a new functional unit (FU) for the multiply-and-accumulate (MAC) operation demonstrated the flexibility of our hardware platform in terms of adding support for newly required functionality, even across application domains: we used an existing protocol processing functional unit as our initial starting point in the design of the MAC FU, a DSP domain functional unit. Also, with a single MAC FU hardware implementation we were able to specify multiple system implementation schemes with different optimization factors for the target application. The implementations were verified with VHDL simulations and synthesis using 0.18 μm CMOS technology. With 16 bit MAC inputs the FU consumed 2.7 kgates, and with 32 bit inputs 7.5 kgates. Overall, the MAC FU was determined to consume about 5-10 % of the total chip area of a typical TACO processor implementation, which we regard highly acceptable.

This work lays foundation for further studies in executing DSP domain operations in parallel with protocol processing tasks on our hardware platform. We are also considering some improvements to the presented MAC FU: first, with the integration of two local memories into the unit, a significant amount of data communication on the interconnection buses would become internal data flow of the MAC FU. Second, we are considering a FU design in which several MAC blocks with local coefficient memories would be interconnected internally. This internal parallelism would provide a significant increase of filtering capacity.

References

1. Corporaal, H.: Microprocessor Architectures - from VLIW to TTA. John Wiley and Sons Ltd., Chichester, West Sussex, England (1998)
2. Tabak, D., Lipovski, G.J.: MOVE architecture in digital controllers. IEEE Transactions on Computers **29** (1980) 180–190
3. Virtanen, S.: A Framework for Rapid Design and Evaluation of Protocol Processors. PhD thesis, Dept. of Information Technology, University of Turku, Finland (2004) http://www.tucs.fi/publications/insight.php?id=phdVirtanen04a/.
4. Hoffmann, A., Kogel, T., Nohl, A., Braun, G., Schliebusch, O., Wahlen, O., Wieferink, A., Meyr, H.: A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language. IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems **20** (2001) 1338–1354
5. Van Praet, J., Lanneer, D., Geurts, W., Goossens, G.: Processor modeling and code selection for retargetable compilation. ACM Transactions on Design Automation of Electronic Systems **6** (2001) 277–307

6. Cichon, G., Robelly, P., Seidel, H., Matúš, E., Bronzel, M., Fettweis, G.: Synchronous transfer architecture (STA). In: Proc. of 3rd and 4th Intl. SAMOS Workshops (LNCS 3133), Springer-Verlag, Berlin, Germany (2004) 343–352
7. Corporaal, H., Hoogerbrugge, J.: Cosynthesis with the MOVE framework. In: Proceedings of the IMACS-IEEE Multiconference on Computational Engineering in Systems Applications (CESA'96), Lille, France (1996) 184–189
8. Heikkinen, J., Sertamo, J., Rautiainen, T., Takala, J.: Design of transport triggered architecture processor for discrete cosine transform. In: Proceedings of the 15th Annual IEEE International ASIC/SOC Conference, Rochester, NY, U.S.A. (2002)
9. Radosavljevic, P., Cavallaro, J.R., de Baynast, A.: ASIP architecture implementation of channel equalization algorithms for MIMO systems in WCDMA downlink. In: Proceedings of the 60th IEEE Vehicular Technology Conference, Los Angeles, CA, U.S.A. (2004)
10. Lilius, J., Truscan, D.: UML-driven TTA-based protocol processor design. In: Proceedings of the 2002 Forum for Design and Specification Languages (FDL'02), Marseille, France (2002)
11. Virtanen, S., Lilius, J.: The TACO protocol processor simulation environment. In: Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES'01), Copenhagen, Denmark (2001) 201–206

Observations on Power-Efficiency Trends in Mobile Communication Devices

Olli Silvén[1] and Kari Jyrkkä[2]

[1] Department of Electrical and Information Engineering,
University of Oulu, Finland
Olli.Silven@ee.oulu.fi

[2] Technology Platforms, Nokia Corporation, Oulu, Finland
Kari.Jyrkka@nokia.com

Abstract. Computing solutions used in mobile communications equipment are essentially the same as those in personal and mainframe computers. The key differences between the implementations are found at the chip level: in mobile devices low leakage silicon technology and lower clock frequency are used. So far, the improvements of the silicon processes in mobile phones have been exploited by software designers to increase functionality and to cut development time, while usage times, and energy efficiency, have been kept at levels that satisfy the customers. In this paper, we explain some of the observed developments.

1 Introduction

During the brief history of GSM mobile phones, the line widths of silicon technologies used for their implementation have decreased from 0.8 μm in the mid 1990's to around 0.13 μm in the early 21st century. In a typical phone, a basic voice call is fully executed in the base band signal processing part, making it a very interesting reference point for comparisons as the application has not changed over the years, not even in the voice call user interface. Nokia gives the "talk-time" and "standby-time" for its phones in the product specifications, measured according to [1] or an earlier similar convention. This enables us to track the impacts of technological changes over time.

Table 1 documents the changes in the worst case talk-times of high volume mobile phones released by Nokia between 1995 and 2003 [2], while Table 2 presents approximate characteristics of CMOS processes during the same period [3], [4], [5]. We make an assumption that the power consumption share of the RF power amplifier was around 50% in 1995. As the energy efficiency of the silicon process has improved, the last phone in our table should have achieved around an 8 hour talk-time with no RF energy efficiency improvements since 1995.

During 1995-2003 the gate counts of the DSP processor cores have increased significantly, but their specified power consumptions have dropped by a factor of 10 [4] from 1mW/MIPS to 0.1mW/MIPS. On the microcontroller side, the energy efficiency of ARM7TMDI, for example, has improved more than 30-fold between 0.35 and 0.13 μm CMOS processes [5]. Obviously, processor developments cannot explain why the energy efficiency of voice calls has not improved, but we need to analyze the system implementations.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 142–151, 2005.

Table 1. Talk-times of three mobile phones from the same manufacturer

Year	Phone model	Talk-time	Stand-by-time	Battery capacity
1995	2110	2 h 40 min	30 h	550 mAh
1998	6110	3 h	270 h	900 mAh
2003	6600	3 h	240 h	850 mAh

Table 2. Past and projected CMOS process development

Design rule (nm)	Supply voltage (V)	Normalized power*delay/gate
800 (1995)	3.15	30
500 (1998)	2.85	12
130 (2003)	1.5	1
60 (2010)	1	0.35

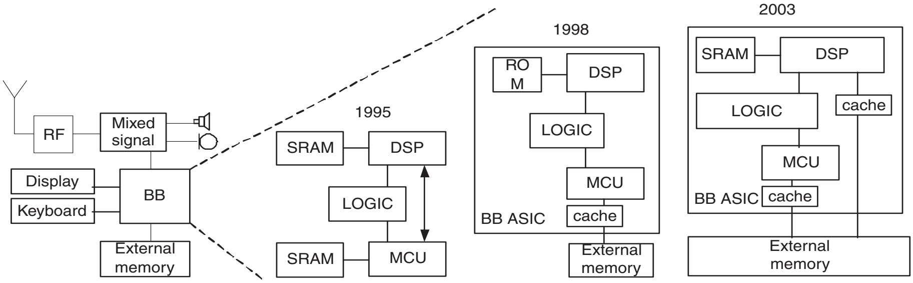


Fig. 1. Typical implementations of mobile phones from 1995 to 2003

Figure 1 depicts streamlined block diagrams of base band solutions of three product generations of GSM mobile phones. The DSP processor runs radio modem layer 1 [6] and the audio codec, whereas the microcontroller (MCU) processes layers 2 and 3 of the radio functionality and takes care of the user interface. During voice calls, both the DSP and MCU are active, while the UI introduces an almost insignificant load.

The base band signal processing ranks second in power consumption after RF during a voice call. Its implementation in 1995 was based on the periodically scheduled software architecture that has almost no overhead, a solution dictated by the performance limitations of the processor used. Hardware accelerators were used without interrupts by relying on their deterministic latencies; this was an inherently efficient and predictable approach. On the other hand, highly skilled programmers, who understood the hardware in detail, were needed. This approach had to be abandoned after the complexity of DSP software and the developer population grew.

In 1998, the DSP and the microcontroller of the user interface were integrated on to the same chip, and the faster DSP processors eliminated some hardware accelerators [7]. Speech quality was enhanced at the cost of additional processing on the DSP, while middleware was introduced on the microcontroller side. The implementation of

2003 employs a pre-emptive operating system in the microcontroller. Basic voice call processing is still on a single DSP processor that now has a multilevel memory system. Improved voice call functionality and lots of other features were supported, and the number of hardware accelerators increased due to higher data rates. The accelerators were synchronized with DSP tasks via interrupts. The architecture is ideal for large development teams, but the new functionalities cause some energy overhead.

The need for better software development processes has increased with the growth in the number of features in the phones. Consequently, the developers have endeavoured to preserve the active usage times of the phones at a constant level (around three hours) and turned the silicon level advances into software engineering benefits.

2 Modern Mobile Computing Tasks

Mobile computing is about to enter an era that requires the integration of wireless wideband data modems, video cameras, and phones into small packages with long battery powered operation times. Even the small size of phones is a design constraint as the sustained heat dissipation should be kept below 3W [8]. In practice, more than the capabilities of laptop PCs is expected using less than 10% of their energy and space, and at a fraction of the price. To understand how the expectations could be met, we briefly consider the characteristics of video encoding and 3GPP signal processing. These have been selected as representatives of soft and hard real time applications, and of differing hardware/software partitioning challenges.

2.1 Video Encoding

Table 3 below illuminates the approximate costs and processing requirements of current common video coding standards when applied to a sequence of 640-by-480 pixel (VGA) images captured at 30 frames/s. The "future standard" has been linearly extrapolated based on those of the past. If a software implementation on an SISD processor is used, MPEG-4 encoding requires the fetching and decoding of at least 200-300 times more instructions than pixel data. This can be used as a basis for comparing implementations on different programmable processor architectures.

Figure 2 illustrates the Mpixels/s per silicon area (mm^2) and power (W) efficiencies of SISD, VLIW, SIMD, and the monolithic accelerator implementations of high quality (>34dB PSNR) MPEG-4 ASP (Advanced Simple Profile) VGA video encoders. Due to the quality requirement the greediest motion estimation algorithms are not applicable. The search area was set to 48-by-48 pixels which fits into the on-chip RAMs of

Table 3. Encoding requirements for 30 frames/s VGA video

video standard	operations/pixel	processing speed (GOPS)
MPEG-4	200-300	2-3
H.264-AVC	600-900	6-10
"future"	2000-3000	20-30

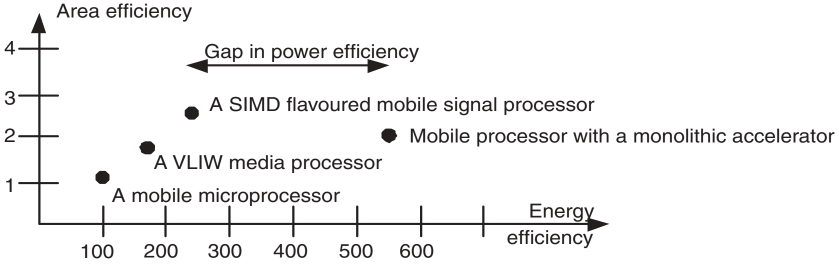


Fig. 2. Area (Mpixels/s/mm^2) and energy efficiencies (Mpixels/s/W) of comparable MPEG-4 encoder implementations

each processor. In the figure the implementations have been normalized to an expected low power 1V 60nm CMOS process. The scaling rule roughly assumes that power consumption is proportional to the supply voltage squared and the design rule, while the die size is proportional to the design rule squared.

The encoder software for the SISD is a commercial one, while for VLIW and SIMD the motion estimators of commercial MPEG-4 ASP codecs were replaced by iterative full search algorithms [9][10]. Unfortunately, we are unable to name the processors in this paper. The monolithic accelerator is an IP block [11] with an ARM926 core.

We notice that around 40mW of power is needed for encoding 10Mpixels/s using the SIMD extended processor, while the monolithic accelerator requires only 16mW. In reality, the efficiency gap is even larger as the data points have been determined using only a single task on each processor. In practice, the processors switch contexts between tasks and serve hardware interrupts, reducing the hit rates of instruction and data caches. This drops the actual processing throughput and energy efficiency.

The sizes of the control units and instruction fetch rates needed for video encoding appear to explain the data points of the programmed solutions. SISD and VLIW have the highest fetch rates, while the SIMD has the lowest one, contributing to energy efficiency. The monolithic accelerator is controlled via a finite state machine, and needs processor services only once every frame. Its hardwired control gives good silicon efficiency: around 5mm^2 is needed for achieving real-time video encoding. The relative size of the SISD processor control unit is the largest in comparison to the execution unit, and results in the worst silicon efficiency for this application.

2.2 3GPP Base Band Signal Processing

The 3GPP base band signal processing chain is an archetypal hard real-time application that is further complicated by the heavy computational requirements shown in Table 4 for the receiver. The values in the table are for solutions employing Turbo-decoding and they do not include chip level decoding and symbol level combining that further increase the processing needs. The requirements of the HSDPA (High Speed Downlink Packet Access) channel characterize currently acute implementation challenges.

Figure 3 shows the organization of the 3GPP receiver processing. The receiver data chain has time critical feedback loops implemented in the software; for instance, the

Table 4. 3GPP receiver requirements for different channel types

Channel type	Data rate	processing speed (GOPS)
Release 99 DCH channel	0.384 Mbit/s	1-2
Release 5 HSDPA channel	14.4 Mbit/s	35-40
“future 3.9G” OFDM channel	100 Mbit/s	210-290

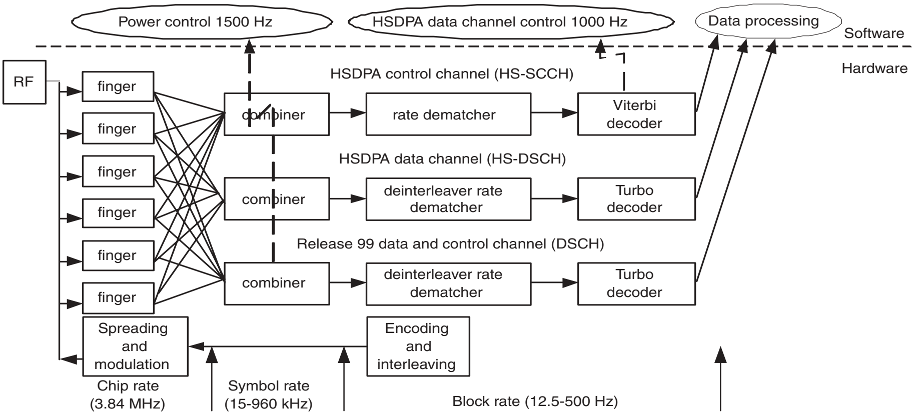


Fig. 3. Receiver for a 3GPP mobile terminal

control channel HS-SCCH is used to control what is received, and when, on the HS-DSCH data channel. Another example is the power control information decoded from “release 99 DSCH” channel that is used to regulate the transmitter power 1500 times per second. The channel code rates, channel codes and interleaving schemes may change anytime, requiring software control for reconfiguring hardware blocks of the receiver, although for clarity this is not indicated in the diagram.

The computing power needs of 3GPP signal processing have so far been satisfied only by hardware at an acceptable energy efficiency level. Software implementations for turbo decoding that meet the speed requirement do exist, but falling short of the energy efficiency needed in phones they are more suitable for base stations [12].

For energy efficiency, battery powered systems have to rely on hardware, while the tight timings demand the employment of fine grained accelerators. A resulting large interrupt load on the control processors is an undesired side effect. Coarser grain hardware accelerators could reduce this overhead, but this approach is risky, if the channel specifications have not been completely frozen, when the development must begin.

3 Analysis of the Observed Development

Based on our understanding, there is no single action that could improve the talk times of mobile phones and usage times of future applications. Rather there are multiple interacting issues for which balanced solutions must be found. In the following, we point out some of the factors considered to be essential.

Voice Call Application. The voice codec in 1995 required around 50% of the operation count of the more recent codec that provide improved voice quality. As a result, the computational cost of a GSM voice call may have even doubled [13], using part of the performance improvement obtained through advances in semiconductor processes. It is likely that the computational costs of voice calls will increase even in the future with advanced features.

Pre-emptive Real-Time Operating Systems. The dominating scheduling principle used in embedded systems is rate monotonic analysis (RMA) that assigns higher static priorities for tasks that execute at higher rates. When the number of tasks is large, utilizing the processor at most up to 69% guarantees that all deadlines are met [14]. If more processor resources are needed, then more advanced analysis is needed to learn whether the scheduling meets the requirements.

Both our application examples are affected by this law. A video encoder, even when fully implemented in software, is seldom the only task in the processor, but shares its resources with a number of others. The 3GPP base band processing chain consists of several tasks due to time critical hardware software interactions.

With RMA, the processor utilization limit alone may demand even 40% higher clock rates than apparently necessary. We cannot return to static cyclic scheduling as it is unsuitable for providing responses for sporadic events within a short fixed time, as required by the newer features of the phones. The use of dynamic priorities and Earliest-Deadline-First or Least Slack algorithm [15] could improve processor utilization over RMA, although at the cost of higher scheduling overheads.

Context Switches and the Cache and Processor Performance. The instruction and data caches of modern processors improve energy efficiency, when they perform as intended. However, when the number of tasks and the frequency of context switches is high, the cache-hit rates may suffer. Both video encoder and 3GPP base band applications may operate in an environment that executes even up to tens of thousands of interrupts and context switches in a second.

Experiments [18] carried out using the MiBench [16] embedded benchmark suite revealed that with a 16kB 4-way set associative instruction cache the hit-rate averaged 78% immediately after context switches and 90% after 1000 instructions, while 96% was reached after the execution of 10000 instructions. The performance impact can be significant. If the processor operates at 150MHz with a 50 ns main memory and an 86% cache hit rate, the execution time of a short task slice (say 2000 instructions) is almost double of the minimum. The time may fluctuate from activation to activation, causing scheduling and throughput complications, and may force to increasing the processor clock rate.

Hardware/Software Interfaces. The designers of mobile phones aim to create common platforms for product families [7]. They define application programming interfaces that remain the same, regardless of system enhancements and changes in hardware/software partitioning. This has made middleware solutions attractive.

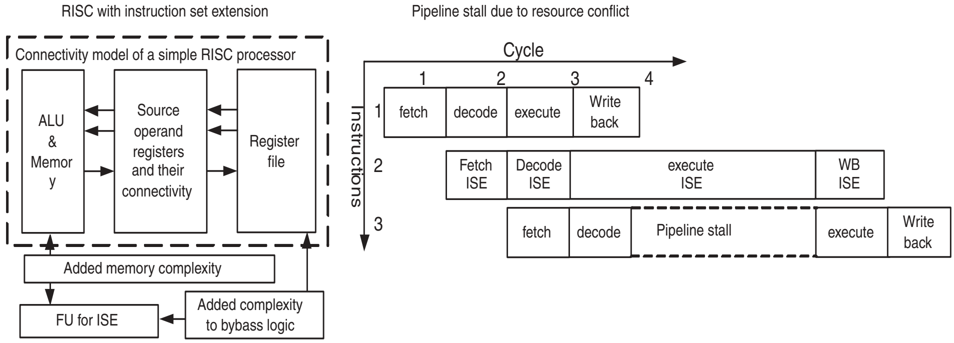

RISC with instruction set extension
Connectivity model of a simple RISC processor
ALU & Memor y
Source operand registers and their connectivity
Register file
Added memory complexity
FU for ISE
Added complexity to bybass logic
Pipeline stall due to resource conflict
Cycle
1
2
3
4
Instructions
1
fetch
decode
execute
Write back
2
Fetch ISE
Decode ISE
execute ISE
WB ISE
3
fetch
decode
Pipeline stall
execute
Write back

Fig. 4. Hardware acceleration via instruction set extension

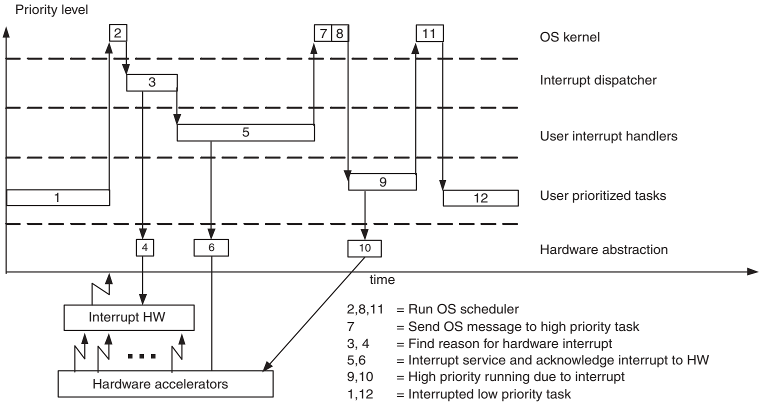

Priority level
2
7
8
11
OS kernel
3
Interrupt dispatcher
5
User interrupt handlers
1
9
12
User prioritized tasks
4
6
10
Hardware abstraction
time
Interrupt HW
...
Hardware accelerators
2,8,11 = Run OS scheduler
7 = Send OS message to high priority task
3, 4 = Find reason for hardware interrupt
5,6 = Interrupt service and acknowledge interrupt to HW
9,10 = High priority running due to interrupt
1,12 = Interrupted low priority task

Fig. 5. Controlling an accelerator interfaced as a peripheral device

Two approaches are available for interfacing hardware accelerators to software. First, a hardware accelerator can be integrated into the system as an extension to the instruction set, as illustrated with Figure 4. The latency of the extension should be in the same range as the standard instructions, or, at most, within a few instruction cycles, otherwise the interrupt response time may suffer. Short latency often implies large gate count and high bus bandwidth needs that reduce the economic viability of the approach.

Second, an accelerator may be used in a peripheral device that generates an interrupt after completing its task. This principle is demonstrated in Figure 5, which also shows the role of middleware in hiding details of hardware. If the code in the middleware is not integrated into the task, calls to middleware functions are likely to reduce the cache hit rate and energy efficiency. Against this fact, it is logical that the monolithic accelerator turned out to be the most energy efficient solution for video encoding in Figure 2. From the point of view the 3GPP base band a key to energy efficient implementation in a given hardware lies in pushing down the latency overheads.

Anything in between 1-2 cycle instruction set extensions and peripheral devices executing thousands of cycles appears to result in inefficient software. If the interrupt latency in the operating system environment is around 300 cycles and 50000 interrupts are generated per second, 10 % of the 150 MHz processor resources are swallowed by the this overhead alone, and on top of this we have middleware costs. This bottleneck area falls between hardware and software, architectures and mechanisms, and systems and components.

Processor and Application Compatibility. Current DSP processor execution units are deeply pipelined to increase instruction execution rates. However, DSP processors are often used as control processors and have to handle large interrupt and context switch loads. The result is a double penalty: The utilization of the pipeline decreases and the control code is inefficient due to the long pipeline. For instance, if a processor has a 10 level pipeline and 1/50 of the instructions are unconditional branches, almost 20 % of the cycles are lost. Improvements offered by the branch prediction capabilities are diluted by the interrupts and context switches.

Summary of Relative Performance Degradations. When the components of the above analysis are combined, they result in an efficiency degradation factor of at least 6. If we add the jitter of code execution times, the factor is easily around 10 or more; and we haven't yet considered programming language and paradigm issues. The result illustrates the traded-off efficiency gains at the processing system level as the approaches in system development have been dictated by the needs of software development.

4 Directions for Research and Development

Looking back to the phone of 1995 in Table 1, we may consider what should have been done to improve energy efficiency at the rate of silicon process improvement. Obviously, due to the choices made, many of the factors that degrade the relative energy efficiency are software related. However, we do not demand changes in software development processes or architectures that are intended to facilitate human effort. So solutions should be sought from the software/hardware interfacing domain, including compilation, and hardware solutions that enable building energy efficient software systems.

To reiterate, the early base band software was effectively multi-threaded, and even simultaneously multi-threaded with hardware accelerators executing parallel threads, without interrupt overhead, as shown in Figure 6. In principle, a suitable compiler could have replaced manual coding in creating the threads, as the hardware accelerators had deterministic latencies. However, interrupts were introduced and later solutions employed additional means to hide the hardware from the programmers.

Having witnessed the past choices, their motivations, and outcomes, we need to ask whether compilers, and hardware support for simultaneous multithreading, could be used to hide implementation details in addition to APIs and middleware. This could cut down the number of interrupts, reduce the number of tasks and context switches, and improve code locality; all improving processor utilization and energy efficiency. Hardware accelerator aware compilation would bridge the software efficiency gap between

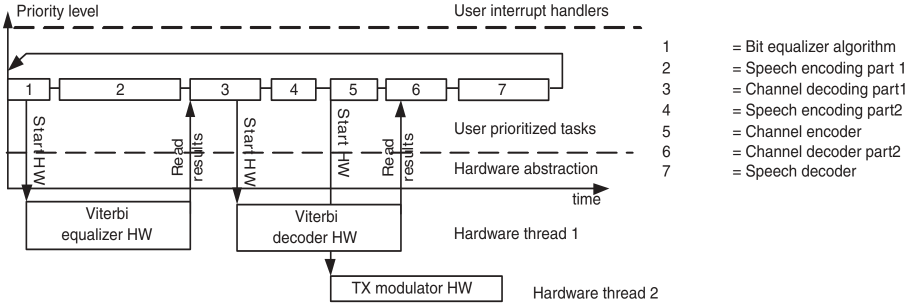


Fig. 6. The execution threads of an early GSM mobile phone

instruction set extensions and peripheral devices, making "medium latency" accelerators attractive. This would help in cutting the instruction fetch and decoding overheads.

A big issue is the paradigm change. Compilers have so far been developed for processor cores; now they would be needed for complete embedded systems. Whenever the platform changes, the compiler would need to be upgraded, while currently the changes are concentrated on the hardware abstraction functionality.

Another approach that could improve energy efficiency is the employing of several small processor cores for controlling hardware accelerators, rather that a single powerful one. This simplifies real-time system design and reduces the penalty from interrupts, context switches and execution time jitter.

5 Summary

The energy efficiency of mobile phones has not improved at the rate that might have been expected from the advances in silicon processes, but it is obviously at a level that satisfies most users. Higher data rates and multimedia applications require significant improvements, and encourage us to reconsidering the ways software is designed, run, and interfaced with hardware.

Significantly improved energy efficiency might be possible even without any changes to hardware by using software solutions that reduce overheads and improve processor utilization. Large savings can be expected from applying architectural approaches that reduce the volume of instructions fetched and decoded.

Acknowledgements

This paper is based on the contributions of numerous people. In particular, we wish to thank Dr. Lauri Pirttiaho and Prof. Yrjö Neuvo, both from the Nokia Corporation.

References

1. GSM Association: TW.09, Battery Life Measurement Technique. (1998)
2. Nokia: Phone models. In: www.nokia.com. (2004)

3. Anis, M., Allam, M., Elmasry, M.: Impact of technology scaling on CMOS logic styles. IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing **49** (2002) 577–588
4. Frantz, G.: Digital Signal Processor Trends. IEEE Micro **20** (2000) 52–59
5. ARM: The ARM foundry program. In: www.arm.com. (2004)
6. 3GPP: TS 05.01, Physical Layer on the Radio Path. In: www.3gpp.org. (2004)
7. Jyrkkä, K., Silven, O., Ali-Yrkkö, O., Heidari, R., Berg, H.: Component-based development of DSP software for mobile communication terminals. Microprocessors and Microsystems **26** (2002) 463–474
8. Neuvo, Y.: Cellular phones as embedded systems. In: Solid-State Circuits Conference. Volume 1. (2004) 32–37
9. Gao, X., Duanmu, C., Zou, C.: A multilevel successive elimination algorithm for block matching motion estimation. IEEE Transactions on Image Processing **9** (2000) 501–504
10. Wang, H., Mersereau, R.: Fast Algorithms for the Estimation of Motion Vectors. IEEE Transactions on Image Processing **8** (1999) 435–438
11. Hantro Products: 5250 VGA encoder. In: www.hantro.com. (2004)
12. Loo, K., Alukaidey, T., Jimaa, S.: High Performance Parallelised 3GPP Turbo Decoder. In: 5^{th} European Personal Mobile Communications Conference. Volume 492., IEE (2003) 337–342
13. Salami, R., Laflamme, C., Bessette, B., Adoul, J.P., Jarvinen, K., , Vainio, J., Kapanen, P., Honkanen, T., Haavisto, P.: Description of GSM enchanced full rate speech codec. In: IEEE International Conference on Communications. Volume 2. (1997) 725–729
14. Klein, M., Ralya, T., B.Pollak, Obenza, R.: A practitioner's handbook for real-time analysis. Kluwer (1993)
15. Spuri, M., Buttazzo, G.: Efficient aperiodic service under earliest deadline scheduling. In: Real-Time Systems Symposium. (1994) 2–11
16. Gathaus, M., Ringenberg, J., Ernst, D., Austen, T., Mudge, T., Brown, R.: MiBench: a free commercially representative embedded benchmark suite. In: IEEE 4^{th} Annual Workshop on Workload Characterization. (2001) 3–14

CORDIC-Augmented Sandbridge Processor for Channel Equalization

Mihai Sima[1], John Glossner[2,3], Daniel Iancu[2], Hua Ye[2], Andrei Iancu[4,2],
and A. Joseph Hoane[2]

[1] University of Victoria, Department of Electrical and Computer Engineering,
P.O. Box 3055 Stn CSC, Victoria, B.C. V8W 3P6, Canada
msima@ece.uvic.ca

[2] Sandbridge Technologies, Inc., 1 North Lexington Avenue, White Plains, NY 10601, USA
{JGlossner, DIancu, HuaYe, AIancu, JHoane}@sandbridgetech.com

[3] Delft University of Technology,
Department of E.E.M.C.S., Delft, The Netherlands

[4] Rochester Institute of Technology, Computer Science Department, Rochester, NY, USA

Abstract. In this paper we analyze an architectural extension for a Sandbridge processor which encompasses a CORDIC functional unit and the associated instructions. Specifically, the first instruction is `CFG_CORDIC` that configure the CORDIC unit in one of the rotation and vectoring modes for circular, linear, and hyperbolic coordinate systems. The second instruction is `RUN_CORDIC` that launches CORDIC operations into execution. As case study, we consider channel estimation and correction of the Orthogonal Frequency Division Multiplexing (OFDM) demodulation. In particular, we propose a scheme to implement OFDM channel correction within the extended instruction set. Preliminary results indicate a performance improvement over the base instruction set architecture of more than 80% for doing channel correction, which translates to an improvement of 50% for the entire channel estimation and correction task.

1 Introduction

A common trade-off in the design of computing engines involves the balance between efficiency and flexibility. Although Application-Specific Integrated Circuits (ASIC's) are highly efficient, they are often not flexible enough to support the variations of today's rapidly evolving standards. On the other hand, DSP processors, although fully programmable, may not achieve the high performance required for future generations of wireless systems. An architectural solution, Application-Specific Instruction set Processors, combines the efficiency of ASIC's and flexibility of DSP's. Typically, ASIP's are heterogenous platforms composed of programmable processor cores and customized hardware modules. Considering the Sandbridge processor [1, 2, 3] and CORDIC algorithm [4, 5], two general questions may be raised:

- What is the influence of a CORDIC functional unit on the performance of a Sandbridge processor?

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 152–161, 2005.

- What are the architectural changes needed for incorporating a CORDIC unit into a Sandbridge processor core?

In order to evaluate the potential performance of the CORDIC-augmented Sandbridge processor, we address as an example channel equalization, which is one of the most challenging baseband wireless algorithms for an Orthogonal Frequency Division Multiplexing (OFDM) demodulation task. In particular, we present the implementation strategy of the channel correction on an ASIP comprising a Sandbridge processor and a CORDIC unit.

The extension of the Sandbridge Instruction Set Architecture (ISA) encompasses a CORDIC functional unit and two associated instructions: `CFG_CORDIC` that configures the CORDIC unit, and `RUN_CORDIC` that launches CORDIC operations into execution. With these instructions, a large number of transcendental functions can be computed in pipelined and SIMD fashion, which translates into a significant reduction in cycle count. In particular, modulus, division, sine, cosine, and arctangent will benefit from CORDIC support when doing channel equalization.

The paper is organized as follows. For background purposes, we outline the CORDIC algorithm in Section 2 and OFDM channel equalization in Section 3. Section 4 describes briefly the architecture of the Sandbridge processor. The architectural extension including CORDIC instructions is introduced in Section 5. The execution scenario of channel correction within the extended instruction set is discussed in Section 6, while experimental results are presented in Section 7. Section 8 completes the paper with some conclusions and closing remarks.

2 Coordinate Rotation Digital Computer

A Givens transformation [6] is a 2-by-2 orthogonal matrix $R(\theta)$ of the form described in Equation (1). It can be observed that multiplication by $R(\theta)$ of a vector $[x,y]^T$ amounts to a counterclockwise rotation of θ radians in plane.

$$R(\theta)\cdot\begin{bmatrix}x\\y\end{bmatrix} \equiv \begin{bmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{bmatrix}\cdot\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}x'\\y'\end{bmatrix} \tag{1}$$

Historically, the Givens transformation has been used in QR factorization [7], since it can zero matrix elements selectively. Clearly, by setting

$$\cos\theta = \frac{x}{\sqrt{x^2+y^2}}, \quad \sin\theta = \frac{y}{\sqrt{x^2+y^2}} \tag{2}$$

it is possible to force the second entry in the vector $[x,y]^T$ to zero:

$$\begin{bmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{bmatrix}\cdot\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}\sqrt{x^2+y^2}\\ 0\end{bmatrix} \tag{3}$$

The Givens transformation is computationally demanding. For example, given an arbitrary angle θ, the direct evaluation of the rotation (Equation (1)) requires four multiplications, two additions, and a large memory storing the cosine and sine tables. Also, finding the angle θ which satisfies the trigonometric Equations (2), translates to a sequence of multiplications, additions, and memory look-up operations if the common Taylor series expansion is employed.

COordinate Rotation DIgital Computer (CORDIC) is an iterative method performing vector rotations by arbitrary angles using only shifts and additions. The main idea is to first split the rotation angle θ into a sequence of subrotations of angles $\theta(n)$, where the rotation for iteration n is

$$\begin{bmatrix} x(n+1) \\ y(n+1) \end{bmatrix} = \begin{bmatrix} \cos\theta(n) & \sin\theta(n) \\ -\sin\theta(n) & \cos\theta(n) \end{bmatrix} \cdot \begin{bmatrix} x(n) \\ y(n) \end{bmatrix} \tag{4}$$

Then, the rotation matrix $R(\theta(n))$ is written as

$$R(\theta(n)) = \cos\theta(n) \cdot \begin{bmatrix} 1 & \tan\theta(n) \\ -\tan\theta(n) & 1 \end{bmatrix} \tag{5}$$

and the rotation angles are restricted so that $\tan\theta(n) = \pm 2^{-n}$. This way, the multiplication by the tangent factor is reduced to simple shift operations.

Arbitrary rotation angles can be obtained by performing a series of successively smaller elementary rotations. If the decision at each iteration, n, is which direction to rotate rather than whether or not to rotate, then the factor $\cos\theta[n]$ becomes a constant for the current iteration (since $\cos\theta[n] = \cos(-\theta[n])$). Then, the product of all these cosine values is also a constant and can be applied anywhere in the system or treated as system processing gain.

The angle of a composite rotation is uniquely defined by the sequence of the directions of the elementary rotations. That sequence can be represented by a decision vector. The set of all possible decision vectors is an angular measurement system based on binary arctangents. Conversions between this angular system and any other can be accomplished using an additional adder-subtractor that accumulates the elementary rotation angles at each iteration. The elementary angles are supplied by a small look-up table (one entry per iteration), or are hardwired, depending on the implementation. The angle accumulator adds a third difference equation to the CORDIC algorithm.

$$z(n+1) = z(n) - d(n)\arctan\left(2^{-n}\right) \tag{6}$$

The CORDIC rotator is operated in one of two modes: rotation or vectoring [4].

- In **rotation mode**, the angle accumulator is initialized with the desired rotation angle. The rotation decision at each iteration is made to diminish the magnitude of the residual angle in the angle accumulator.
- In **vectoring mode**, the CORDIC unit rotates the input vector through whatever angle is necessary to align the result vector with the x axis. The result of the vectoring operation is a rotation angle and the scaled magnitude of the original vector (the x component of the result).

Using CORDIC, a large number of transcendental functions, e.g., polar to cartesian or cartesian to polar transformations, can be calculated with the latency of a serial multiplication. By providing an additional parameter, the basic CORDIC method can be generalized to perform rotations in a linear or hyperbolic coordinate system [5], thus providing a more powerful tool for function evaluation. Of particular importance for this paper is CORDIC operating in vectoring mode in the linear coordinate system, since it provides a method for evaluating ratios.

3 OFDM Channel Equalization

Due to its robustness to multi-path propagation conditions and support for high data rates, coded Orthogonal Frequency Division Multiplexing (OFDM) has become one of the most popular modulation techniques for indoor and outdoor broadband wireless data transmission [8]. OFDM has been adopted in many wireless worldwide standards such as wireless LAN 802.11a/g, HIPERLAN/2, Digital Audio Broadcasting (DAB), Digital Video Broadcasting Terrestrial (DVB-T), Digital Video Broadcasting for Handheld (DVB-H), WirelessMAN 802.16, and Broadband Wireless Access.

Consider DVB-T/H [9]: the transmitted signal is organized in frames, and each frame consists of 68 OFDM symbols. Each symbol contains both data and reference information, and is constituted by a set of 6817 carriers in the 8K mode and 1705 carriers in the 2K mode. Due to phase and amplitude variations in the channel transfer function, received complex data symbols appear not only rotated in the complex domain, but also attenuated or enhanced. Under these conditions the amplitude and phase of each carrier is distorted. If the receiver is to coherently demodulate the signal, it needs to equalise the phase and amplitude of each carrier. This process, which is known as Channel Equalisation, is comprised of an estimation phase and a correction phase.

Channel estimators usually employ pilot information as a point of reference. A fading channel requires constant tracking, so pilot information has to be transmitted continuously. An efficient way of allowing a continuously updated channel estimate is to transmit pilot symbols instead of data at certain locations of the OFDM time-frequency lattice. This technique is referred to as Pilot-Assisted Transmission (PAT).

Most OFDM receivers, such as DVB-T/H [9], are PAT systems. The channel estimation and correction is performed for the current OFDM symbol using a set of pilot carriers [10, 11, 12]. In the following, we describe at a high level the channel estimation we used for our DVB-T/H implementation. For each OFDM symbol, the scattered pilots are spaced 12 carriers apart. The first step is the estimate of the Channel Transfer Function (CTF) samples on the scattered pilot positions for the current OFDM symbol. The second step in channel estimation is to perform interpolation in time domain for three virtual pilot groups [8]. The third step in channel estimation is to perform interpolation in frequency domain. This is done to estimate the remaining two CTF samples in between each virtual pilot pairs for the current symbol. Once CTF samples for the carriers are estimated, channel correction can be readily performed to get the corrected received carriers that are ready to be fed to further processing such as QAM demapping and TPS decoding [9].

The correction algorithm is a so called *derotation*. Assuming the channel estimation yields an error vector $e_I + je_Q$ for a particular carrier, the corrected vector for that particular carrier is obtained by the complex division

$$c_I + jc_Q = \frac{r_I + jr_Q}{e_I + je_Q} = \frac{(r_I + jr_Q)(e_I - je_Q)}{e_I^2 + e_Q^2} \tag{7}$$

where c_I and c_Q are the corrected values for the real and imaginary part of a particular carrier, and r_I and r_Q are the real and imaginary parts of the received carrier. Each carrier must go through the computationally-intensive derotation process described in Equation 7.

4 Overview of the Sandbridge Processor

In this section we describe the most important issues of the Sandbridge architecture and microarchitecture. In particular, our emphasis will be on the multi-threading capability and SIMD-style Vector Unit.

4.1 Sandbridge Processor

Sandbridge Technologies has designed a multithreaded processor capable of executing DSP, embedded control, and Java code in a single compound instruction set optimized for handset radio applications [1,2,3]. The Sandbridge Sandblaster design overcomes the deficiencies of previous approaches by providing substantial parallelism and throughput for high-performance DSP applications, while maintaining fast interrupt response, high-level language programmability, and low power dissipation.

The Sandbridge processor [1,2,3] is partitioned into three units; an instruction fetch and branch unit, an integer and load/store unit, and a SIMD-style vector unit. The design utilizes a unique combination of techniques including hardware support for multiple threads, SIMD vector processing, and instruction set support for Java code. Program memory is conserved through the use of powerful compounded instructions that may issue multiple operations per cycle. The resulting combination provides for efficient execution of DSP, control, and Java code. The instructions to speed up CORDIC operations are executed in the Sandbridge Vector Unit described in Subsection 4.4.

4.2 Sandbridge Pipeline

The pipelines are different for various operations as shown in Figure 1. The Load/Store (Ld/St) pipeline has nine stages. The integer and load/store unit has two execute stages for Arithmetic and Logic Unit (ALU) instructions and three execute stages for integer multiplication (I_MUL) instructions. A *Wait* stage for the ALU and I_MUL instructions causes these instructions to read from the general-purpose register file one cycle later than Ld/St instructions. This helps reduce the number of register file read ports. The vector multiplication (V_MUL) has four execute stages – two for multiplication and two for addition. It should be noted that once an instruction from a particular thread enters the pipeline, it runs to completion. It is also guaranteed to write back its result before the next instruction from the same thread reads the result.

4.3 Sandbridge Multithreading

The Sandblaster architecture supports multiple concurrent program execution by the use of hardware thread units. Multiple copies (e.g., banks and/or modules) of memory are

Ld/St	Inst Dec	RF Read	Agen	XFer	Int Ext	Mem 0	Mem 1	Mem 2	WB
ALU	Inst Dec	Wait	RF Read	Exec 1	Exec 2	XFer	WB		
I_Mul	Inst Dec	Wait	RF Read	Exec 1	Exec 2	Exec 3	XFer	WB	
V_Mul	Inst Dec	VRF Read	Mpy1	Mpy2	Add1	Add2	XFer	VRF WB	

Fig. 1. Sandbridge pipeline

available for each thread to access. The Sandblaster processor uses the Token Triggered Threading (T^3) form of interleaved multithreading [2], in which each thread is allowed to simultaneously execute an instruction, but only one thread may issue an instruction on a cycle boundary. The microarchitecture currently supports up to eight concurrent hardware threads. Multi-threading effectively hides true dependencies which typically occur in connection with long-latency operations.

V_Mul	Inst Dec	VRF Read	Mpy1	Mpy2	Add1	Add2	XFer	WB			
V_Mul									Inst Dec	VRF Read	Mpy1

Fig. 2. Two consecutive Vector Multiply instructions that issue from the same thread

4.4 The Vector Processing Unit

The Vector Processing Unit (VPU) consists mainly of four Vector Processing Elements (VPEs), which perform arithmetic and logic operations in SIMD fashion on 16-bit, 32-bit, and 40-bit fixed-point data types. High-speed 64-bit data busses allow each PE to load or store 16 bits of data each cycle in SIMD fashion. Support for SIMD execution significantly reduces code size, as well as power consumption, since multiple sets of data elements are processed with a single instruction [13].

Most SIMD vector instructions go through eight pipeline stages. For example, a vector MAC (V_MAC) instruction goes through the following stages: Instruction Decode, Vector Register File (VRF) Read, Mpy1, Mpy2, Add1, Add2, Transfer, and Write Back. The Transfer stage is needed due to the long wiring delay between the bottom of the VPU and the VRF. Since there are eight cycles between when consecutive instructions issue from the same thread, results from one instruction in a thread are guaranteed to have written their results back to the VRF by the time the next instruction in the same thread is ready to read them. Thus, the long pipeline latency of the VPEs is effectively hidden, and no data dependency checking or bypass hardware is needed. This is illustrated in Figure 2, where two consecutive vector multiply instructions issue from the same thread. Even if there is a data dependency between the two instructions, there is no need to stall the second instruction, since the first instruction has completed the Write Back stage before the second instruction enters the VRF Read stage.

5 An Architectural Extension for Sandbridge Processor

The instructions investigated are CFG_CORDIC that configures the CORDIC unit in one of the execution modes (rotation, vectoring) and one of the coordinate systems (circular, linear, hyperbolic), and RUN_CORDIC which launches the configured CORDIC operation. Assuming that 16-bit precision is needed (it is usually the case in OFDM demodulation), then the CORDIC algorithm reads in two 16-bit arguments and produces two 16-bit results. If not all the CORDIC iterations can be performed by a single RUN_CORDIC call, then the angle and iteration number must be saved between successive RUN_CORDIC calls.

The proposed RUN_CORDIC instruction is a vector instruction that goes through eight pipeline stages; that is, the execution itself has a latency of 4 thread cycles. The CORDIC functional unit can perform 2 CORDIC iterations in a thread cycle (two additions and two shifts), and is shared by four SIMD units. Consequently, RUN_CORDIC will execute 8 times, i.e., it will take up 8 instruction cycles, for a 16-bit precision, and will perform 4 conversions in SIMD style.

This is the result where the CORDIC unit is added to the vector unit by adding one adder, one shifter, a comparator and some control logic to the existing pipeline. Deploying an autonomous CORDIC unit for each SIMD unit of each thread will translate into both a memory bandwidth problem and a hardware problem, i.e., too much added hardware (32 adders, 32 shifters, 32 comparators), and the operands cannot be fetched from, or the results cannot be stored back to memory anyway.

The CORDIC instructions are defined as follows.

- CFG_CORDIC
  for(i=0; i<4; i++) {
  - Read 8 bits of configuration data
  - Configure the CORDIC unit:
    - Mode (1 bit): rotation or vectoring
    - Coordinate system (2 bits): circular, linear, or hyperbolic
    - Iteration Identifier (5 bits): ranges from 0 to 31

  }

- RUN_CORDIC
  for(i=0; i<4; i++) {
  - Reads in the first 32-bit vector register packing:
    - 16-bit modulus and 16-bit angle for rotation mode
    - 16-bit x-value and 16-bit y-value for vectoring mode
  - Reads in the second 32-bit vector register storing:
    - 16-bit angle for vectoring mode
  - Performs two CORDIC iterations (two additions and shiftings)
  - Writes back one 32-bit vector register packing:
    - 16-bit x-value and 16-bit y-value for rotation mode
    - 16-bit modulus and 16-bit angle for vectoring mode

  }

6 Channel Correction Execution Scenario

As mentioned in Section 6, the channel correction involves essentially a complex division. The strategy we implemented for channel correction is to express complex numbers in trigonometric form and to use the CORDIC algorithm to perform the computation. Briefly, this strategy can be summarized as follows:

1. Express the numerator of Equation 7 in trigonometric form by using CORDIC (vectoring mode, circular rotations).
2. Express also the denominator of Equation 7 in trigonometric form by using CORDIC (vectoring mode, circular rotations).
3. Perform the division using CORDIC (vectoring mode, linear rotations).
4. Express the result back in algebric form also using CORDIC (rotation mode, circular rotations)

Assume 16-bit precision: it requires 16 vector instructions to complete a CORDIC rotation. Therefore, 64 vector instructions are used to perform the derotation. The computing performance according to this scenario has been evaluated for a pure software solution and also when CORDIC operation benefits from customized instruction set. The experimental results are presented in the next section.

7 Experimental Results

The Sandbridge integer ALU (non-vectorized) takes $9 \times 16 = 144$ instructions to do a rotate. This implies $144 \times 4 = 576$ instructions are needed to do a derotate. The Vectorized loop takes $8 \times 16 = 128$ instructions to do 4 rotates, which implies $128 \times 4 = 512$ instructions for 4 derotates. The ALU takes 1 instruction longer because the ALU has no MAC instruction and must use conditional jump. If the ALU had a CORDIC instruction, it would take 0.5×16 instructions to do 1 rotate. If the Vector Unit had a CORDIC instruction, it would take 0.5×16 instructions to do 4 rotates. The figures are presented in Table 1.

Although CORDIC is essentially a sequential algorithm (it can compute a number of functions in a serial way, one bit per iteration), it has the very important property of being vectorizable and pipelineable. This explains the very good performance provided

Table 1. Derotation figures per carrier

Implementation style	1 thread (instructions)	8 threads (cycles)
ALU using emulated division	96	96
ALU using emulated CORDIC	$4 \times 9 \times 16 = 576$	576
ALU using hardware CORDIC	$4 \times 16 = 64$	64
Vector using emulated CORDIC	$4 \times (8 \times 16)/4 = 128$	128
Vector using hardware CORDIC	$(4 \times 16)/4 = 16$	16

by the 4-way CORDIC unit when doing derotation (16 cycles per carrier) over the non-vectorized ALU solution (96 cycles per carrier).

Experiments which have been carried out on a cycle-accurate simulator provide for the following numerical figures. Channel equalization total cycle count per OFDM symbol is 319634 cycles, out of which the complex division (that is, the derotation) in Equation 7 counts for 209388 cycles. Given the fact that the CORDIC-based solution provides for a cycle count reduction of $(96 - 16) \times 100/96 = 83\%$, the global improvement for channel estimation is $0.83 \times 209388 \times 100/319634 = 54\%$. Given the fact that Sandbridge is a multi-threaded DSP-oriented processor, such an improvement within wireless processing domain indicates that extending the Sandbridge instruction set with CORDIC instructions is a promising approach.

8 Conclusions

We have proposed an architectural extension for the Sandbridge processor which encompasses a CORDIC functional unit and the associated instructions: `CFG_CORDIC` and `RUN_CORDIC`. Configuring the CORDIC unit in one of the two modes and three coordinate systems is performed under the command of the `CFG_CORDIC`, while the `RUN_CORDIC` instruction launches into execution CORDIC operations. Preliminary results indicate a performance improvement over the base instruction set architecture of more than 80% for doing channel correction, which translates to an improvement of more than 50% for the entire channel estimation and correction task. As future work, we intend to address the entire DVB-T processing chain and to evaluate the overall system improvement from the CORDIC-augmented Sandbridge processor.

References

1. Glossner, J.C., Hokenek, E., Moudgill, M.: Multithreaded Processor for Software Defined Radio. In: Proceedings of the 2002 Software Defined Radio Technical Conference. Volume I., San Diego, California (2002) 195–199
2. Schulte, M.J., Glossner, J.C., Mamidi, S., Moudgill, M., Vassiliadis, S.: A Low-Power Multithreaded Processor for Baseband Communication Systems. In Pimentel, A.D., Vassiliadis, S., eds.: Proceedings of the Third and Fourth International Annual Workshops on Systems, Architectures, MOdeling, and Simulation (SAMOS). Volume 3133 of Lecture Notes in Computer Science., Samos, Greece, Springer (2004) 393–402
3. Glossner, J.C., Schulte, M.J., Moudgill, M., Iancu, D., Jinturkar, S., Raja, T., Nacer, G., Vassiliadis, S.: Sandblaster Low-Power Multithreaded SDR Baseband Processor. In: Proceedings of the 3rd Workshop on Applications Specific Processors (WASP'04), Stockholm, Sweden (2004) 53–58
4. Volder, J.E.: The CORDIC trigonometric computing technique. IRE Transactions on Electronic Computers **EC-8** (1959) 330–334
5. Walther, J.: A unified algorithm for elementary functions. In: Proceedings of the Spring Joint Computer Conference of the American Federation of Information Processing Societies (AFIPS). Volume 38., Arlington, Virginia, U.S.A., AFIPS Press (1971) 379–385
6. Golub, G.H., van Loan, C.F.: Matrix Computations. 3rd edn. The Johns Hopkins University Press (1996)

7. Strang, G.: Introduction to Linear Algebra. 3rd edn. Wellesley-Cambridge Press (2003)
8. van Nee, R.D., Prasad, R., eds.: OFDM for Wireless Multimedia Communications. Artech House Publishers (2000)
9. European Telecommunications Standards Institute: Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for digital terrestrial television (2004)
10. Speth, M., Fechtel, S., Fock, G., Meyr, H.: Optimum Receiver Design for OFDM-Based Broadband Transmission – Part II: A Case Study. IEEE Transactions on Communications **49** (2001) 571–578
11. Frescura, F., Pielmeier, S., Reali, G., Baruffa, G., Cacopardi, S.: DSP-Based OFDM Demodulator and Equalizer for Professional DVB-T Receivers. IEEE Transactions on Broadcasting **45** (1998) 323–332
12. Tong, L., Sadler, B.M., Dong, M.: Pilot-Assisted Wireless Transmissions: General Model, Design Criteria, and Signal Processing. IEEE Signal Processing Magazine **21** (2004) 12–25
13. Sebot, J., Drach, N.: SIMD ISA Extensions: Reducing Power Consumption on a Superscalar Processor for Multimedia Applications. In: IEEE Symposium on Low-Power and High-Speed Chips (Cool Chips) IV, Tokyo, Japan (2001)

Power-Aware Branch Logic: A Hardware Based Technique for Filtering Access to Branch Logic*

Sunghoon Shim[1], Jong Wook Kwak[1], Cheol Hong Kim[1], and Sung Tae Jhang[2],
and Chu Shik Jhon[1]

[1] School of Electrical Engineering and Computer Science,
Seoul National University, Seoul, Korea
{shshim, leoniss, kimch, csjhon}@panda.snu.ac.kr
[2] Department of Computer Science, The University of Suwon,
Suwon, Gyeonggi-do, Korea
stjhang@suwon.ac.kr

Abstract. In this paper, we propose a power-aware branch logic for high performance embedded processors by filtering access to BTB and branch predictor. The proposed scheme reduces the energy consumed in BTB and branch predictor. For reducing the energy consumption in the BTB and the branch predictor, we present an aggressive hardware-based scheme that reduces the number of access to the BTB and the branch predictor. Moreover, compared with general branch logic, the proposed branch logic has no performance degradation. This scheme reduces the number of access to the BTB and the branch predictor by 21% - 50% and reduces the energy consumption in the BTB and the branch predictor by 15% - 41%.

1 Introduction

Design of embedded processors has focused on low power consumption than high performance compared with general-purpose processors, since the embedded processors have been mainly used for application-specific devices. However, the exclusive growth of multi-functional hand-held devices like PDAs and some mobile multimedia devices requires not only low power consumption but also high computing power in the embedded processors.

In order to achieve high performance, several schemes have been proposed in the processor design. The parallelizing program is one of them. The technique has been generally adapted in many processor architectures including embedded processors and general-purpose processor. In the technique, branch is a significant impact on the program parallelization. As a means of mitigating this effect, several techniques have been suggested for the processor design.

Branch Target Buffer(BTB) and branch predictor are the techniques for alleviating of the branch effect[1]. More complicated branch predictor and larger size of BTB

* This work was supported by the Brain Korea 21 Project.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 162–171, 2005.

have been used, since the accuracy of the branch prediction affects the program parallelization and performance. In addition to more complex branch predictor and larger BTB, it is considered to access the branch logic(we define that a branch logic is composed of a BTB and a branch predictor in this paper) in the fetch stage of pipeline for more performance improvement. However, more complex branch predictor, larger BTB and accessing the branch logic every cycle cause significant energy consumption in the branch logic. The energy consumption in the branch logic accounts for over 10% of the total processor energy consumption[2].

In general, branch instructions occupy over 10% of total instructions in a program[2]. The branch logic is needed for only branch. It means that the accessing the branch logic of 90% isn't needed. Therefore over 90% of the energy consumption for accessing the branch logic is dissipated, since the branch logic is accessed every cycle, regardless of whether the instruction that is being fetched is a branch or not. If an instruction that is fetched next cycle becomes known whether the instruction is a branch or not before being fetched, the access to the branch logic can be reduced dramatically. Consequently, the energy consumption in the branch logic can decrease corresponding on the reduced number of access to the branch logic.

In this paper, we propose an power-aware branch logic that filters access to branch logic without any performance degradation compared with general branch logic. The proposed scheme can identify non-branch instructions before the instruction is fetched. It is based on not compiler-help, but hardware. Our proposed scheme can filter the accesses to the branch logic, resulting in reducing energy consumption in the branch logic.

The rest of this paper is organized as follows. Section 2 describes the previous works. Section 3 presents our proposed scheme. Section 4 describes our simulation methodology and shows detailed experimental results. Finally, Section 5 concludes this paper.

2 Related Works

2.1 Branch Target Buffer and Branch Predictor

Control instructions such as branch instructions cause the performance degradation of programs, since the right sequence of instruction execution isn't known until the branch instruction is executed. Branch logic(branch target buffer(BTB) and branch predictor) has proposed to alleviate the performance loss due to the branch. The technique is based on predicting branch result. The branch logic is looked up by Program Counter(PC) in the fetch stage. If a matching of entry happen in the BTB, the target address in the BTB is used for fetching next instruction according to the result of the branch prediction. The branch predictor determines the direction of the next instruction. If the prediction is taken, the next instruction is the instruction of target address. If the prediction is not-taken, the next instruction to be fetched is the fall-through instruction.

2.2 Reducing Energy of Branch Prediction

Many works have been proposed for low power branch prediction. Pipeline gating was proposed by Manne et al. as an efficient technique to prevent mis-speculated instructions from entering the pipeline and wasting energy while imposing only a negligible performance loss[3]. Chaver et al. proposed a methodology to reduce the energy consumption of the branch predictor by characterizing prediction demand using profiling and dynamically adjusting predictor resources accordingly[4]. In the technique, components of the hybrid direction predictor are disabled and the branch target buffer is resized for low power consumption. Monchiero et al. presented power-aware branch prediction techniques based on a compiler hint mechanism to filter the access to the branch predictor[5].

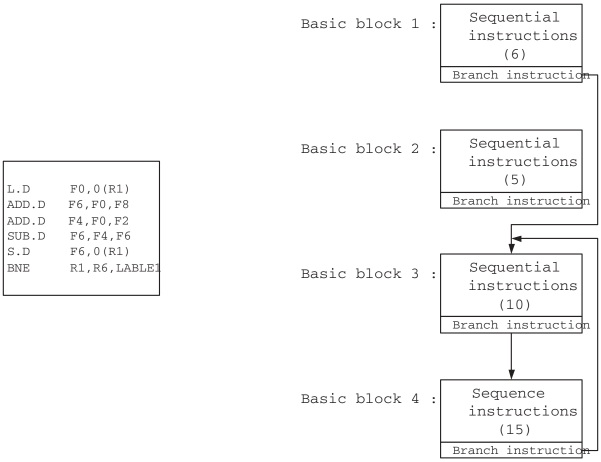
To reduce the power consumption in branch predictors, several processors exploit the pre-decode bits to detect whether the instruction is a branch or not, resulting in selective access to the branch predictor only when the instruction is a branch. In this case, the access to the branch predictor should be preceded by the access to the instruction cache. However, if the fetch stage is timing-critical, the sequential access incurs significant performance loss. PPD(Prediction Probe Detector) scheme is proposed to mitigate the access time[2]. The scheme uses pre-decode bits to entirely eliminate unnecessary accesses to branch logic.

Petrov et al. presented a scheme for low-power BTB for application-specific embedded processors based on compiler hints[6]. The technique utilizes application-specific information regarding the control-flow structure of the program major loops. The information is extracted during compile time and transferred to the Application-Customizable Branch Target Buffer(ACBTB). The scheme needs to profile each application before executed and extra load parts in each application for loading information about branch produced by compiler. Furthermore, the data in ACBTB can’t be modified during runtime.

3 Power-Aware Branch Logic

3.1 Branch Functional Overview

The basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit, as shown Fig.1 (a) [7]. Figure 1 (b) shows an example of general control flow of a program. The program is composed of some basic blocks. The number in each basic block represents the total number of instructions in each basic block. For example, the total number of instructions is 10 in basic block 3. At the end of basic block 1, the next instruction to be fetched is determined by the result of the branch prediction. If the branch in basic block 1 is taken-branch, the next instruction to be fetched is the first instruction in basic block 3. Then, the rest of instructions in the basic block 3 are fetched in sequence until the end of the basic block 3. However, if the branch in basic block 1 is not-taken-branch, the next instruction to be fetched is the first instruction in basic block 2. Then, the rest of instructions in basic block 2 are fetched and executed in sequence until the end of the basic block 2. In case of the taken-branch in basic 1, the number of access to the branch logic for basic block 3 is 10, since the number of instructions in basic block 3 is 10 and the branch logic is accessed whenever instructions in basic block 3 are fetched. However, fetching of the rest 9 instruction except for an branch instruction in the basic block 3 doesn’t need to access the branch logic. The unnecessary 9 accesses to the branch logic are caused by no information for identifying branch instruction. If there is a scheme for identifying branch instructions, it isn't needed to access the branch logic every cycle. Accordingly, the energy to access the branch logic can be reduced dramatically.



```
L.D      F0,0(R1)
ADD.D    F6,F0,F8
ADD.D    F4,F0,F2
SUB.D    F6,F4,F6
S.D      F6,0(R1)
BNE      R1,R6,LABLE1
```

Fig. 1. (a) example of a basic block and (b) control of basic blocks

3.2 Design of Branch Logic

Figure 2 shows an overview of the proposed hardware-based power-aware branch logic that filters the accesses to the branch logic. For filtering access, our power-aware branch logic uses the size of basic block to be fetched. The proposed branch logic is composed of four parts. One is general branch logic, the others are additional three parts, 1)filtering access to branch logic, 2)two tables for storing size of next basic block, 3)computing size of basic block.

The A(filtering Access to Branch Logic) in Fig.2 is a part for recognizing whether an instruction to be fetched is a branch or not. For identifying branch instruction, A part uses the size of basic block to be fetched. A basic block is composed of sequential non-branch instructions and a branch as shown in Fig. 1 (a). Before fetching next basic block, the size of the next basic block is loaded into count register, and whenever a instruction in the next basic block is fetched, the value in the count register is decreased by 1, if the value is 0, the instruction which will be fetched is identified as a branch instruction. Accordingly, the value in the count register is compared with 0 every cycle before the branch logic is accessed. The value in the count register which is 0 denotes that the instruction to be fetched is a branch instruction, or there is no information about the instruction. Only when the value in the count register is 0, the branch logic is accessed such as normal branch logic, and new size of next basic block which will be fetched is loaded from the Taken Count Table(TC Table) or the Not-Taken Count Table(NTC Table) into the count register according to result of the branch prediction. After executing the branch instruction, if the branch prediction is incorrect, the value in the count is reset to 0.

The B(Both tables for count value) in Fig. 2 is a part for storing size(the number of instructions) of the next basic block to be fetched. The part is composed of two small tables. One is the Not-Taken Count Table(NTC Table) for preserving the size(the number of instructions) of the fall through basic block which will be fetched when the branch prediction is not-taken. The other is the Taken Count Table(TC Table) for preserving the size(the number of instructions) of the target basic block which will be fetched when the branch prediction is taken. The number of entry in the two tables is equal to that of entry in the BTB. The size of an entry is the same as that of the count register. When the BTB is hit, data from the TC Table and the NTC Table is read, and then only one value is transferred into the count register according to the result of branch prediction. If the branch prediction is taken, the value from the TC Table is moved to the count register. Otherwise, the value from the NTC Table is moved to the count register. The value of all entry in the two tables is initialized to 0. Whenever the BTB and the branch predictor is updated, an entry in the TC table or NTC Table corresponding to the updated entry of the BTB is set with value from part C. The BTB is updated only when the branch is taken, however TC Table is updated when the branch which makes the BTB updating is not-taken, and NTC table is updated when the branch is not-taken.

The C(Computing part for value of count) in Fig. 2 is a part for computing the size(the number of instructions) of the basic block. The part is constituted by 5 subcomponents. First subcomponent is a Last Committed Branch Address Register(LCBAR). The LCBAR is a register for storing the address of last committed branch(Nth branch). Second subcomponent is composed of a Previous Committed Branch Target Address Register(PCBTAR) and a subtractor for computing difference between value of LCBAR and value of PCBTAR. The PCBTAR is a register for storing target address of previ-

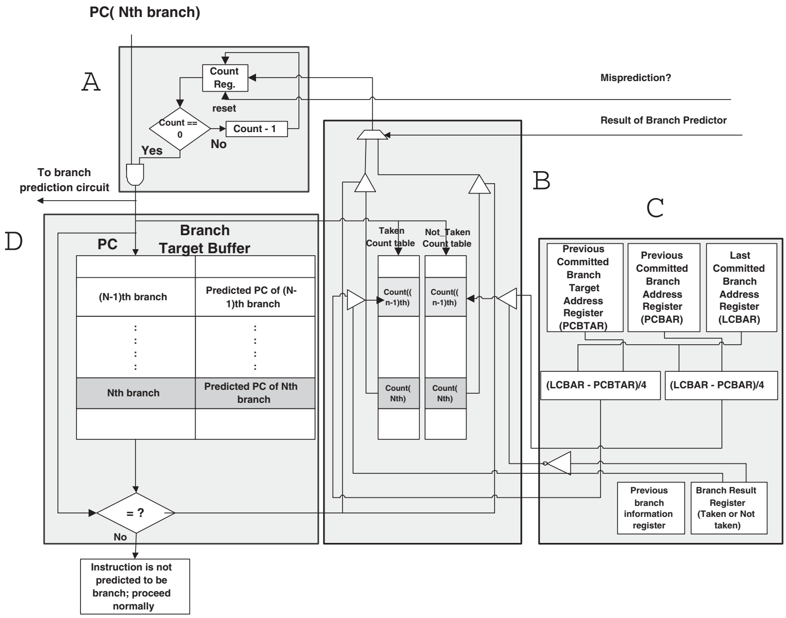
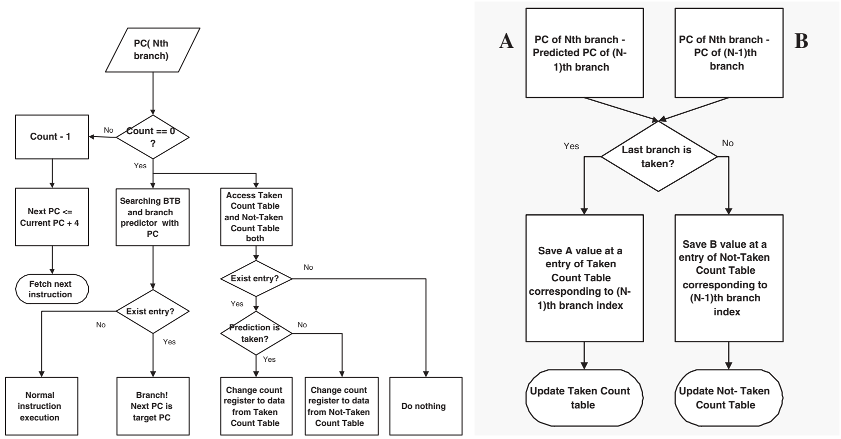

PC(Nth branch)
A
Count Reg.
reset
Count == 0
Count - 1
Yes
No
To branch prediction circuit
Misprediction?
Result of Branch Predictor
B
C
D
Branch Target Buffer
PC
(N-1)th branch
Predicted PC of (N-1)th branch
:
Nth branch
Predicted PC of Nth branch
= ?
No
Instruction is not predicted to be branch; proceed normally
Taken Count table
Not_Taken Count table
Count((n-1)th)
Count((n-1)th)
Count(Nth)
Count(Nth)
Previous Committed Branch Target Address Register (PCBTAR)
Previous Committed Branch Address Register (PCBAR)
Last Committed Branch Address Register (LCBAR)
(LCBAR - PCBTAR)/4
(LCBAR - PCBAR)/4
Previous branch information register
Branch Result Register (Taken or Not taken)

Fig. 2. An overview of the power-aware Branch Logic architecture

ous committed branch((N-1)th branch). The difference means the size(the number of instructions) of the target basic block of previous committed branch((N-1)th branch). Third subcomponent is composed of a Previous Committed Branch Address Register(PCBAR) and a subtractor for computing difference between value of LCBAR and value of PCBAR. The PCBAR is a register for storing address of previous committed branch((N-1)th branch). The difference means the size(the number of instructions) of the fall through basic block of previous committed branch((N-1)th branch). When the difference is over the bit-size of an entry of the TC Table or NTC Table, only value corresponding to the bit-size of an entry in the Tables is moved to the entry. Forth subcomponent is a Previous Committed Branch Information Register(PCBIR). The value in the PCBIR is used to store the result of computation in the TC Table or the NTC Table for previous committed branch((N-1)th branch). Last subcomponent is a register for result of previous committed branch((N-1)th branch). According to the result, the value for storing in TC or NTC Table is determined. If the result is true(it means that previous committed branch((N-1)th branch) is a taken-branch), the difference(size of target basic block) between value of LCBAR and value of PCBTAR is stored in the TC Table. Otherwise, the difference(the size of fall-through basic block) between value of LCBAR and value of PCBAR is stored in the NTC Table.


PC(Nth branch)
Count == 0 ?
No
Yes
Count - 1
Next PC <= Current PC + 4
Fetch next instruction
Searching BTB and branch predictor with PC
Exist entry?
No
Yes
Normal instruction execution
Branch! Next PC is target PC
Access Taken Count Table and Not-Taken Count Table both
Exist entry?
No
Yes
Prediction is taken?
No
Yes
Change count register to data from Taken Count Table
Change count register to data from Not-Taken Count Table
Do nothing
A
PC of Nth branch - Predicted PC of (N-1)th branch
B
PC of Nth branch - PC of (N-1)th branch
Last branch is taken?
Yes
No
Save A value at a entry of Taken Count Table corresponding to (N-1)th branch index
Save B value at a entry of Not-Taken Count Table corresponding to (N-1)th branch index
Update Taken Count table
Update Not- Taken Count Table

(a) (b)

Fig. 3. Flow-chart for filtering access to branch logic

Figure 3 (a) depicts the flow control of the proposed scheme for low power consumption. Figure 3 (a) explains the mechanism for filtering accesses to the branch logic. Figure 3 (b) describes the process of computing count and updating count table. The number of instructions for TC Table and NTC Table are calculated during the commit phase in pipeline stage.

4 Experimental Results

4.1 Experimental Framework

Our simulation is based on the Wattch simulator[8] originated from SimpleScalar[9]. We modified the Wattch simulator for our scheme. The processor is in-order 2 issues processor. Table 1 shows the base configuration for our simulation systems. This model is similar to the next ARM processor[10].

We simulated 20 applications from SPEC2000 benchmarks, 10 benchmarks from SpecINT 2000 and 10 benchmarks from SpecFP 2000[11]. We simulated both base scheme and our proposed scheme. The base scheme is a conventional BTB and gshare-branch predictor scheme. We simulated the two count tables having various entry size(bits) from 10 bits to 3 bits and count register.

4.2 Access to Branch Logic

Figure 4 depicts the reduced rate of access to the branch logic compared with normal branch logic. In normal branch logic, whenever instructions are fetched, the branch logic is accessed. However, in our proposed scheme, many accesses to the branch logic is filtered by identifying branch instructions.

In Fig. 4, the bars of each benchmark show the reduced access rate various size of the count register. The sizes are from 3 bits(leftmost bar) to 10 bits(rightmost bar). The sizes represent the size of the count register and the size of an entry in the TC and the NTC Table. The maximum number of instructions to be filtered is determined according to the bit size. If the size is 3 bits, 7 is the maximum number of instructions not to access the branch logic, even though the number of instructions in the basic block is 15. Similarly, 10 bits means that 1024 is the maximum number of instructions

Table 1. System configuration parameters

Processor Core	
Issue width	2 issues per cycle, inorder issue 2 Int ALU, 1 Int mult/div 2 FP ALU, 1 FP mult
Memory Hierarchy	
L1 I-cache	32KB, 2-way, 32 byte blocks, 1 cycle latency
L1 D-cache	32KB, 4-way, 32 byte blocks, 1 cycle latency, WB
L2 cache	unified, 4-way, 256 KB, 64 byte blocks, 8 cycle latency
Memory latency	64 cycle latency
Branch Predictor	
Branch predictor	Gshare, 4096 entry in the level 1
Branch Target Buffer	512 entry, 1-way
Added Part for Filtering Access to BTB	
Table for count value	512 entry in Taken Count Table 512 entry in Not-Taken Count Table 3 - 10 bit size of an entry

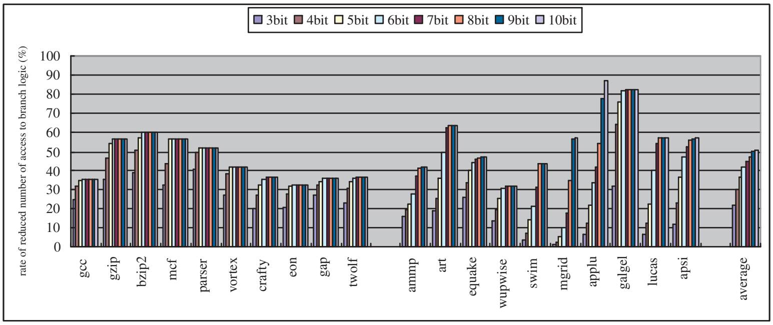


Fig. 4. Rate of reduced number of access to branch logic

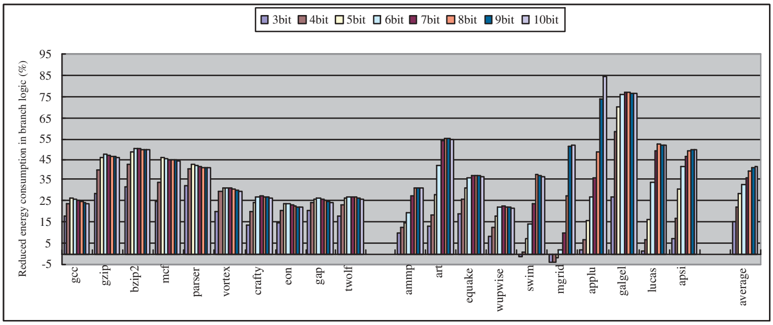


Fig. 5. Reduced energy consumption in branch logic

that don't need to access the branch logic. As shown Fig. 4, the accesses to the branch logic are reduced by 21% - 50%, on average. The figure shows that large bit size can reduce more accesses than small bit size. However, in almost benchmarks, using large bit size over a certain bit has no impact on reduced rate of accesses, since the certain bit size can cover the size of each basic block in the benchmarks. For examples, in bzip2, the reduced access rate increases until 6 bits, and then there are no change over 7 bits. It is caused by the size of the largest basic block in the bzip2. 72 are the number of instructions in the largest basic block, and 72 can be indicated by 7 bits. Another benchmarks have similar characteristic.

In Fig. 4, the results of SpecFP(right part) is similar to results of SpecINT(left part). The size of basic blocks in the SpecFP are relatively larger than that of the SpecINT, from 129 to 807. Accordingly, the difference of reduced access rate in SpecFP between using large bit and using small bit is larger than that of SpecINT.

4.3 Reduction in Branch Logic Energy

Figure 5 shows the reduced energy consumption of the proposed branch logic. In our proposed scheme, the extra energy consumption is added to the normal branch logic system. The extra energy is mainly consumed by the TC Table, the NTC table. However, the energy consumption is very smaller than the reduced energy consumption. In Fig. 5, the energy consumption of the branch logic is reduced by 15% - 41%, on average.

Using large bit size can reduce more accesses to the branch logic than using small bit size as shown in Fig. 4. However, larger bits need higher energy consumption(i.e. 3 bit vs. 10 bit), since it requires large size of TC Table and NTC Table. In the SpecINT, the size(numbers of instructions) of each basic block are mainly between 33 and 128. Accordingly, using 8, 9, 10 bits size is required more energy than using 5, 6, 7 bit size, since the size of TC Table and NTC Table become large, and using 3, 4 bit size reduces less access than using 5, 6, 7. Consequently, most SpecINT benchmarks in left side of Fig. 5 show that using middle size bits, i.e. 5, 6 and 7 bit size, can reduce more energy consumption of the branch logic than using 3, 4, 8, 9 and 10 bit size.

The SpecFP benchmarks in right side of Fig. 5 shows different aspects of results compared with the SpecINT benchmarks as shown in left side of Fig. 5. The figure depicts that using large bit size(9, 10 bit) reduces more energy consumption than using small bit size. It is caused by characteristic of SpecFP. In general, a basic block in SpecFP has more instructions than that in the SpecINT. Therefore, the gap of energy reduction rate in SpecFP between using small bit size and using large bit size is bigger than that of energy reduction rate in the SpecINT.

5 Conclusions

The energy consumption in a branch logic is getting higher, since high performance, program parallelizing and high accuracy of branch prediction are needed for processor design. In this paper, we proposed power-aware branch logic by filtering access to branch logic in high performance embedded processor. For filtering access to the branch logic, our proposed scheme uses information about identifying branch instructions. The information is size of next basic block to be fetched. The size is extracted by hardware during runtime, it doesn't need compiler hints or compiler help. The proposed scheme can reduce a lot of accesses to the branch logic and also reduce energy consumption in the branch logic without any performance degradation. The accesses to the branch logic are reduced by 21% - 50%. The energy consumption is reduced by 15% - 41% in the branch logic.

References

1. Perleberg, C.H., Smith, A.J.: Branch target buffer design and optimization. IEEE transactions on computers **42** (1993) 396–412
2. Parikh, D., Skadron, K., Zhang, Y., Barcella, M., Stan, M.R.: Power issues related to branch prediction. In: In proceedings of the 8th international symposium on High-Performance Computer Architecture. (2002) 233–246
3. Manne, S., Klauser, A., Grunwald, D.: Pipeline gating: speculation control for energy reduction. In: In Proceedings of the 25th Annual International Symposium on Computer Architecture. (1998) 132–141
4. Chaver, D., Pinuel, L., Prieto, M., Tirado, F., Huang, M.C.: Branch prediction on demand: an energy-efficient solution. In: In proceedings of the International Symposium on Low Power Electrinics and Design '03. (2003) 390–395

5. Monchiero, M., Palermo, G., Sami, M., Silvano, C., Zaccaria, V., Zafalon, R.: Power-aware branch prediction techniques: a compiler-hints based approach for vliw processors. In: ACM Great Lakes Symposium on VLSI 2004. (2004) 440–443
6. Petrov, P., Orailoglu, A.: Low-power branch target buffer for application-specific embedded processors. In: Proceedings of the Euromicro Symposium on Digital System Design. (2003) 158–165
7. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. 3 edn. Morgan Kaufmann (2003)
8. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: A framework for architectural-level power analysis and optimizations. In: In Proceedings of the 27th Annual International Symposium on Computer Architecture. (2000) 83–94
9. Burger, D.C., Austin, T.M.: The simplescalar tool set,version 2.0. Computer Architecture News **25** (1997) 13–25
10. (2004) http://www.arm.com/miscPDFs/6871.pdf.
11. Standard Performance Evaluation Corporation: SPEC CPU2000 Benchmarks. (2000) http://www.specbench.org/osg/cpu2000.

Exploiting Intra-function Correlation with the Global History Stack

Fei Gao and Suleyman Sair

Department of Electrical and Computer Engineering,
NC State University, Raleigh, NC 27695, USA
{fgao, ssair}@ece.ncsu.edu

Abstract. The demand for more computation power in high-end embedded systems has put embedded processors on parallel evolution track as the RISC processors. Caches and deeper pipelines are standard features on recent embedded microprocessors. As a result of this, some of the performance penalties associated with branch instructions in RISC processors are becoming more prevalent in these processors. As is the case in RISC architectures, designers have turned to dynamic branch prediction to alleviate this problem. Global correlating branch predictors take advantage of the influence past branches have on future ones. The conditional branch outcomes are recorded in a global history register (*GHR*). Based on the hypothesis that most correlation is among intra-function branches, we provide a detailed analysis of the *Global History Stack (GHS)* in this paper. The GHS saves the global history in the return address stack when a call instruction is executed. Following the subsequent return, the history is restored from the stack. In addition, to preserve the correlation between the callee branches and the caller branches following the call instruction, we save a few of the history bits coming from the end of the callee's execution. We also investigate saving the GHR of a function in the Branch Target Buffer (BTB) when it returns so that it can be restored when that function is called again. Our results show that these techniques improve the accuracy of several global history based prediction schemes by 4% on average. Consequently, performance improvements as high as 13% are attained.

1 Introduction

With an ever growing number of uses and applications, the computation demand on embedded systems has reached new heights. To meet these challenges, embedded microprocessor designers started introducing microarchitectural features such as caches and pipelining which are not common in the cost-conscious embedded domain. Newly released high-end processors routinely feature these techniques [1, 2, 3]. One of the major performance bottlenecks due to pipelining is the branch misprediction penalty. When considering the fact that these high end processors can execute multiple instructions every cycle, branch mispredictions can potentially induce a considerable waste of execution resources (both cycles and power). Furthermore, if the branch depends on a long latency instruction such as a divide or a load that misses in the data cache, the branch resolution time can grow even longer. As a result, accurate branch prediction is the *key* factor in eliminating this penalty.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 172–181, 2005.

Global predictors exploit the influence of past branch instructions on future ones. In a typical global predictor, a *Global History Register (GHR)* establishes the correlation between branches. The GHR records conditional branch outcomes and becomes part of the index into the branch prediction table. Consequently the number of history bits held in the GHR is of critical importance in the accuracy of a correlating branch predictor [4]. This is especially true for global predictors that have special features to eliminate negative interference in the prediction tables [4, 5, 6, 7, 8, 9].

One aspect of applications that reduce the effective GHR width is function calls. When a parent function calls one of its children functions, the GHR of the parent is overwritten by the branches in the child. By the time we return back to the parent, the branch outcomes that the ensuing parent branches are most likely correlated with have been wiped out of the GHR. Calder et al. found that on average, C functions execute 133.6 instructions over a wide range of applications [10]. Furthermore, they report that one out of every 9.3 instructions in these programs is a conditional branch. From these numbers we can deduce that approximately 15 conditional branches execute each time a function call takes place. Considering the fact that most branch predictor implementations have GHRs with 16 or fewer bits, a function call overwrites all but one bit of the branch outcomes belonging to the parent. This problem is exacerbated in embedded microprocessors. First, embedded applications feature many more function calls than typical desktop applications. Additionally, embedded processors have relatively shorter (8-bits or less) GHRs.

This paper proposes saving the GHR of the parent function in the return address stack when a function call occurs. Subsequently we restore the GHR using the value in the return address stack upon returning from the callee. This ensures that the conditional branch outcomes generated before the call are still available in the GHR after returning from the call. In case a few of the trailing branches in the callee function influence branches in the caller, we also investigate saving their values instead of clobbering them. Additionally, we evaluate storing the GHR of a function in the Branch Target Buffer (BTB) when it returns and restoring the GHR when the function is called again to look for branch correlation between its consecutive instances.

The contributions of this paper are:

- Analyze the sources of correlation among branches separated by function calls
- Examine correlation between branches in different instances of a function
- Investigate several low cost, low overhead techniques to exploit these two types of correlation
- Provide a detailed performance analysis of these designs and quantify their impact on branch prediction accuracy

The rest of this paper is organized as follows. We show a couple of code examples illustrating why intra-function correlation is hindered by function calls and how correlation across function instances exists in Section 2. Section 3 presents some background into global branch prediction. Section 4 introduces the different mechanisms we employ to preserve and improve intra-function correlation. Next, we describe our simulation methodology and the details of the chosen benchmarks in Section 5. We then discuss the impact of the proposed schemes on prediction accuracy and performance in Section 6. Finally, Section 7 summarizes our findings and concludes.

2 Motivation

There are many techniques that take advantage of correlation among branches [11, 12, 13, 14, 15]. In these schemes, the predictor stores past branch outcomes either in a single register or in a separate entry of a PC-indexed table depending on whether the predictor exploits local (i.e. among instances of the same branch) or global (i.e. among different branches) correlation. Function calls do not pose a problem for local correlation because each branch gets its own entry in the table and aside from interference, the history is not perturbed by other branches. Global prediction accuracy however suffers when the history register contents is lost across function calls. Figure 1 lists a code segment from the SPEC'95 program `gcc`. This loop is within the function `copy_rtx_if_shared` in the source file `emit-rtl.c`. The code includes two loops with many function calls (including recursive ones) inside. The callee functions overwrite the GHR bits from `copy_rtx_if_shared` and makes it very hard to correctly predict the `for` loops which are normally very predictable. A means of recovering the GHR upon returning from a call addresses this problem.

```
for (i = 0; i < GET_RTX_LENGTH(code); i++)
  {
    switch (*format_ptr++)
     {
      case 'e':
        XEXP(x, i) = copy_rtx_if_shared (XEXP(x, i));
        break;
      case 'E':
        if (XVEC(x, i) != NULL)
         {
          register int j;
          int len = XVECLEN(x, i);

          if (copied && len > 0)
            XVEC(x, i) = gen_rtvec_v(len, &XVECEXP(x, i, 0));
          for (j = 0; j < len; j++)
            XVECEXP(x, i, j) = copy_rtx_if_shared(XVECEXP (x, i, j));
         }
        break;
      }
  }
```

Fig. 1. Example code segment from the SPEC'95 `gcc` benchmark. This code exemplifies one problem our paper aims to solve. There are two loops executing many function calls which completely overwrite the GHR bits of the caller

```
while (count>0)
 {
         c2=getranchar(c1,ran2());
         text_buffer[bufindex++]=c2 ;
         ...
      c1=c2;
         count--;
 }
```

Fig. 2. Loop from SPEC'95 benchmark `compress`. `getranchar` function benefits from correlation among branches from different invocations of a function

Meanwhile, recursive calls also exhibit an interesting behavior. Even though there may be correlation among instances of the function, in this particular case recursion hinders the prediction process. Each recursive call results in a different loop trip count depending on the length of the subexpressions being analyzed. This means there are many GHR patterns at the end of these recursions and they each train completely different entries in the predictor table, potentially causing destructive aliasing. Another source of distant correlation is displayed in the `getranchar` function in Fig. 2. `getranchar` takes a character and a random number as arguments and returns another character. This loop is inside the function `fill_text_buffer` in source file `harness.c`. The `while` loop forces the character generated in the previous iteration to be used as the starting point in the current iteration. This results in branch outcomes of the previous iteration affecting the branches in the current one. We can preserve this correlation if we can remember the pattern in the GHR at the end of the previous iteration.

3 Related Work

Recognizing the fact that the outcome of some branches depends heavily on other recent branches, Yeh and Patt proposed the *GAg scheme* which uses a global history register to index into the predictor table instead of the PC [16]. The global history consists of a shift register updated with the outcome of each committed branch instruction. The global history may not provide enough information to distinguish the current branch however. For those branch instructions that do not benefit from global correlation, this results in a worse performance than the bimodal predictor. To overcome this, McFarling proposed the *Gshare predictor* hashing the branch PC with the global history to form the index [12]. He found that the exclusive OR of the branch address with the global history gives more information than either component alone when used as an index.

In general, branch predictor entries are not tagged. Consequently this results in entries being shared by multiple branches. This is referred to as *aliasing* or *interference*. When two branches with opposing biases alias, this results in poor branch prediction accuracy for both branches. To eliminate this *negative interference* a *Skew predictor* was proposed [5, 6]. This scheme uses three two-bit counter tables indexed with different hash functions. The intuition is that even if two branches alias in one table, they will hopefully map to separate entries in the other two tables. Other "de-aliased" branch predictors include the *Bi-mode* predictor [9], *Agree* predictor [7] and the *YAGS* predictor [8]. In principal, the idea is to separate the predictor tables for mostly taken and mostly not-taken branches so that any aliasing will result in neutral interference. The Alpha EV8 processor implements an aggressive branch predictor in 2Bc-gskew [6, 4]. As most recent commercially implemented predictors, 2Bc-gskew is a hybrid predictor. It combines bimodal prediction with a "de-aliased" skew predictor to further improve its accuracy.

In addition to designing new global predictors, another way to improve prediction accuracy is to enhance the correlation among branches. Nair proposed dynamic path-based branch correlation [17] which forms a history of past branches addresses instead of their outcomes. This information is able to represent the execution path resulting in more accurate prediction. In [18], Jacobson et al. proposed a path-based next trace predictor to form sequences of traces to index a trace cache with. They also introduce the return history stack (RHS). The operation of a return address stack (RAS) requires information on an instruction-level granularity. Since traces do not entail this much detail, they can not utilize a RAS. The primary focus of RHS is compensating for the absence of the RAS by improving the predictability of branches after a return. It uses a similar stack like architecture to our GHS to restore global history contents following a return. In this paper, we provide an in depth analysis of the effects of intra-function correlation in the context of superscalar branch prediction in much more detail. Furthermore, we propose an architectural extension to the BTB to store GHR values at the end of functions and we evaluate the implications of preserving correlation across instances of the same function.

Another scheme that potentially increases the effective correlation distance is [15]. In this technique Thomas et al. analyze the dynamic data-flow between instructions to find the producers (direct and indirect) of the values used in branch instructions. Next, they determine the branches that these producer instructions are control dependent on.

This process yields a set of branches called *affectors* that directly or indirectly *affect* the computation of values consumed by branch instructions. Hence a particular branch is most likely to be correlated with its affector branches.

4 Proposed Architectural Extensions

There are two aspects of intra-function that we can preserve: 1) the GHR of the parent function through function calls, 2) the GHR of any function across different invocations. We will now detail the mechanisms that we utilize to achieve these goals.

4.1 Global History Stack

In order to preserve the branch outcomes before a function call, we need a temporary storage to save the GHR. The return address stack (RAS) fits this purpose perfectly as it saves the return address under the exact conditions that we want the GHR maintained. We can simply extend each RAS entry with a field to hold the pre-call GHR. This field would be populated together with the return address field on a call and freed when the RAS entry is popped following a return.

Let us illustrate the operation of the GHS with an example. Consider the code segment shown in Fig. 3, which shows three functions. The instruction marked as 1 causes the call instruction at the return address ($0x12004CA8$) and the current GHR (the 10-bit binary value 1000111101) to be pushed on to the next available entry in the RAS. This operation is marked with circle 1 in Fig. 3. Subsequently when the second call instruction is executed, a similar series of events take place and the return address, history pair of ($0x12005F50$,0100101110) is inserted into the RAS (operation 2 in Fig. 3). In the function starting at address $0x12006684$, there are two conditional branches of which the first one is taken and the second one is not taken. With this assumption, the GHR has the value 0010111010 by the time we reach the return instruction as shown in Fig. 3. When the return instruction executes, we pop the RAS, obtaining the predicted target of the return as well as the new value of the GHR (operation 3 in Fig. 3).

For functions that are called from multiple call sites different paths lead to the call site. This results in a different GHR pattern each time the function is called from a different call site. Especially when functions are relatively short, such as in C++ programs where on average there are 5 conditional branches per function invocation [10], this hinders the predictability of these branches because separate two-bit counters need to be trained for each different path. One approach to resolve this issue is clearing the GHR each time a function is called. We call this *zeroing*.

Note that restoring the GHR to its pre-call state can actually hurt prediction accuracy if branches subsequent to returning from a function are dependent on some of the branches in the callee. In this scenario, conditional branch instructions executed prior to returning from the callee influence the return value of the function. Hence upon returning from the callee, when the caller uses the return value as a predicate to conditional branch instructions, it would find it beneficial to correlate with the branches that the return value is control dependent on. These branches are called the *affector* branches in [15]. For this purpose, we evaluate preserving a few of the most recent branch outcomes of the callee function when we restore the GHR using the value in the RAS.

```
①  0x12004CA4     call  0x12005F48
   0x12004CA8     ...

   0x12005F48     ...
   ...
②  0x12006F4C     call  0x12006684
   0x12006F50     ...

   0x12006684     ldq_u   t1,  0(a0)
   0x12016688     bne     t1,  0x12016690 // taken
   0x1201668c     addq    v0,  0x4,  t0
   0x12016690     ldq     t0,  8(v0)
   0x12016694     addq    v0,  0x8,  v0
   0x12016698     cmpbge  zero,  t0,  t1
   0x1201669c     beq     t1,  0x12016680 // not taken
   0x120166A4     subq    zero,  t1,  t2
③  0x120166A4     ret
```

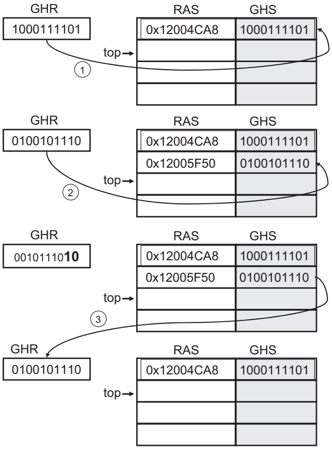


Fig. 3. Operation of GHS: Code example and the corresponding GHR and GHS values when the code executed

4.2 Preserving GHR Values Across Function Invocations

Zeroing solves the problem of having different GHR values when a function is called from different call sites. However, by clearing out the GHR on each call, it prevents taking advantage of any correlation. We can improve the predictability of these branches if we *remember* the GHR from the past invocation of that function.

As shown in Section 2 (recall Fig. 2), there are various examples of functions that have loops and other conditional nests that can benefit from the history of previous instances of these branches. The intermittent branch instructions between two invocations of a function clobber the GHR however. This results in lost correlation opportunities. To mitigate this problem we propose saving the GHR value in the BTB before returning from a function. The next time the same function is called, we can restore the value from the BTB and utilize the past history of branches in the function.

Recall the two `for` loops in Fig. 1. Remembering the histories from one instance of the function to the next can improve the predictability of these loops by training separate predictor entries to reflect the steady state behavior as well as the termination of these loops. To this end, we add an *old history* field to the BTB. When we return from a child function, we update the BTB entry corresponding to the parent's call instruction with the current GHR value. We can simply obtain the PC of the call instruction by subtracting one (actually the size of one instruction) from the return address. The next time that call instruction is fetched, we check the BTB. On a tag match, if the call bit in the BTB is set, we will fill the GHR using the value in the old history field of the BTB. Note that this technique associates the previous history with a particular call site, not a function. In other words, if a function is called from multiple places, the history that will be reloaded into the GHR will be coming from the previous instance of the function when called from the same particular call site. We can store the beginning PC of a function when we enter it in a temporary register and use that PC to store the GHR before returning. But that would potentially introduce non-branch instructions into the BTB and reduce its effective size. Instead of creating a separate table for saving function GHR instances, we chose the simpler approach of adding a field to the BTB and maintaining regular BTB semantics.

5 Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [19], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. Latency values for the caches and register files were obtained using CACTI [20] for a *130nm* process technology.

Table 1. Baseline misprediction rates

Benchmarks	com	gcc95	go	ijpeg	li	vor95	crafty	gcc2K	twolf	vor2K
% Mispred	18.9	10.6	27.6	13.2	6.1	5.4	10	12.2	15.2	5.4

To perform our evaluation, results were collected for 10 of the SPEC95 and SPEC2000 integer benchmarks that were similar to typical embedded programs and had higher than a 5% branch misprediction rate with a 4K entry gshare predictor (see Table 1. These were `compress`, `gcc`, `go`, `ijpeg`, `li`, and `vortex` from SPEC95, and `gcc`, `twolf`, and `crafty` and `vortex` from SPEC2000 suites respectively. The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). We skipped the initialization of each program by skipping 500 million instructions (except for `gcc` from SPEC'95 which executes for fewer than 500 million instructions) and simulated for 100 million committed instructions. All benchmarks were simulated using the *ref* inputs. We evaluate the effectiveness of the proposed intra-function correlation enhancements on a gshare predictor. The predictor has 4K entries and uses an 8-bit global history. The performance analysis models a next generation embedded processor similar to the configuration of ARM11 [1]. The processor can execute 4 instructions every clock cycle. It includes a 10 entry return address stack and a 512 entry BTB.

6 Results

We present the results of our experiments on the efficacy of the proposed techniques in this section. All the results except for the baseline configuration utilize zeroing. In these results GHS refers to adding a history field to the RAS to restore the GHR across function calls. `GHS+r6` is overwriting 6 most significant bits of the GHR (i.e. retaining the last 2 bits) in addition to GHS. Results for the combination of GHS and adding an old history field to the BTB in order to remember the GHR value from the previous

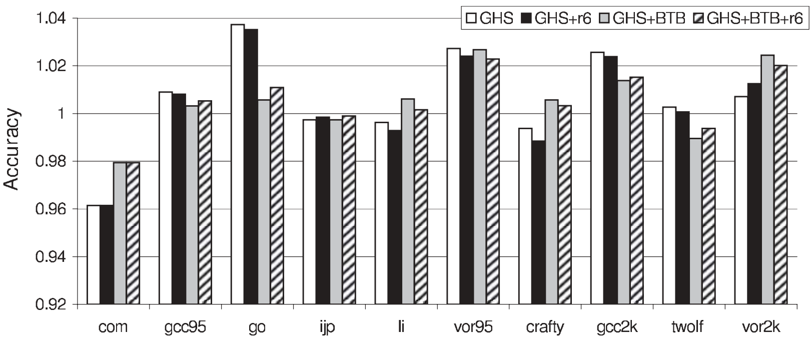


Fig. 4. Normalized branch prediction accuracy of different intra-function correlation techniques used in conjunction with a gshare predictor. The gshare predictor has a 4K entry table and 8 bit global history. The results are normalized to the baseline case

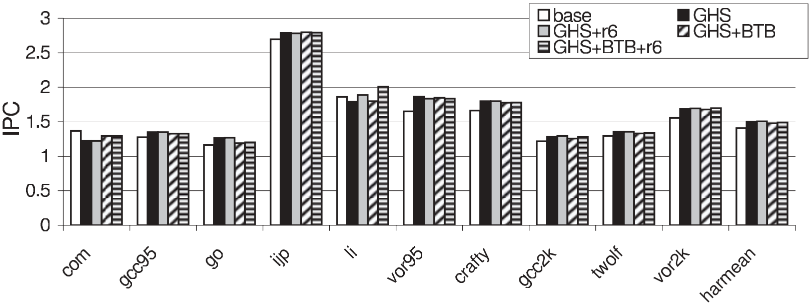


Fig. 5. Performance results for a gshare predictor with 4K entries and a 8-bit global history for different intra-function correlation schemes

iteration of a function are listed as GHS+BTB. And finally adding retaining to this last combination is referred to as GHS+BTB+r6. For the BTB cases if the beginning PC of the function is not found in the BTB, the GHR is initialized to 0.

We show the prediction accuracy for each benchmark normalized to the baseline predictor without the intra-function correlation enhancements. Figure 4 quantifies how effective we are in exploiting intra-function correlation for a 4K entry gshare predictor with 8 bits of history. We can observe several trends in this figure. Applications such as gcc, go and twolf perform better without the addition of the history field to the BTB. In these benchmarks the GHR patterns show great variation from one invocation of a function to another, rendering restoration of the GHR from the BTB ineffective. In this case zeroing proves better as in each instance of the function the history will start from 0. Contrast this behavior to those of compress, li, crafty and vortex. In these applications, the GHS+BTB combinations outperform GHS alone. Even though compress gets a big boost from the addition of BTB extensions, it still performs worse than the baseline gshare predictor. This is primarily attributable to situations where a conditional branch is directly data dependent on the return value of a function. Since we restore the GHR to its pre-call state, the opportunity to exploit this correlation is lost. We experimented with longer retaining values for `compress` and found that prediction accuracy is restored back to the same level as the baseline predictor. Despite the techniques working well on some programs and worse on others, the average misprediction rate is reduced by 4%. To attain best possible results, we are currently investigating adaptive application of GHS techniques where the compiler determines which functions have access to the GHS.

We also measured the performance of these different schemes on a next generation embedded processor with a gshare predictor. Figure 5 displays the number of committed instructions per cycle (IPC) for a 4K entry gshare predictor with 8 bits of history. The last set of bars represent the harmonic mean of the IPC for all benchmarks. On average the `GHS+BTB+r6` configuration improves performance by 5%. Individually, `vortex` enjoys a performance boost of 13% from the GHS while `compress` suffers a slowdown of 11% in the worst case.

7 Conclusions

Global branch prediction is a powerful tool in tolerating control related stalls in a pipelined processor. Whether as a stand-alone predictor or as part of a hybrid predictor, it is extensively used in current processor designs. In global prediction, correlation is established through a Global History Register (GHR). The limited size of the GHR causes callee functions to overwrite the GHR of the parent function, thrashing correlation. Furthermore, branches from previous invocations of a function can influence the direction branches in the current instance will take. Remembering the branch history when the function was executed the last time can improve branch prediction accuracy by cutting down on training time through providing a steady starting point for each invocation of a function.

In this paper, we proposed several intra-function correlation preservation mechanisms. The first of these, the Global History Stack (GHS), saves the history information of the parent when it calls another function into the return address stack (RAS). When the callee finishes and returns, we pop the history value from the RAS. We introduced *zeroing* and *retaining* as means of providing stable starting points for functions and preserving callee-to-caller correlation. Finally, to promote the inherent control dependence across branches in consecutive instances of a function, we proposed saving the global history register (GHR) at the time of the return in the BTB, only to be restored when that function is called again.

The proposed techniques provide, on average, a 4% reduction in misprediction rate. Absolute reduction in misprediction rates is as high as 3% in the case of `go`. Performance improvements as high as 13% (for `vortex95`) are observed. In general, the fact that some programs benefit from our techniques while others do not encourages future research in application specific use of the GHS and the BTB extensions. One compiler solution is identifying branches that benefit from intra-function correlation and apply these techniques only to those functions. This can be done via profiling as was done in [21]. Runtime techniques similar to the methods used in [15] can also help, where we analyze the register and control dependencies to determine correlating branches.

References

1. ARM Ltd.: ARM1136 Technical Reference Manual, Version r0p2. (2004) http://www.arm.com.
2. Intel Corp.: The Intel(R) XScale(TM) Microarchitecture Technical Summary. (2000) http://www.intel.com/design/intelxscale/.
3. Analog Devices Inc.: Analog Devices Blackfin Processor Data Sheet. (2005) http://www.analog.com/processors/processors/blackfin/.
4. Seznec, A., Felix, S., Krishnan, V., Sazeides, Y.: Design tradeoffs for the Alpha EV8 conditional branch predictor. In: Proc. Ann. Int. Symp. Comput. Architecture. (2002) 295–306
5. Michaud, P., Seznec, A., Uhlig, R.: Trading conflict and capacity aliasing in conditional branch predictors. In: Proc. Ann. Int. Symp. Comput. Architecture. (1997)
6. Seznec, A., Michaud, P.: De-aliased hybrid branch predictors. Technical Report RR-3618, Inria (1999)
7. Sprangle, E., Chappell, R., Alsup, M., Patt, Y.: The agree predictor: A mechanism for reducing negative branch history interference. In: Proc. Ann. Int. Symp. Comput. Architecture. (1997) 284–291
8. Eden, A.N., Mudge, T.: The YAGS branch prediction scheme. In: Proc. Ann. ACM/IEEE Int. Symp. Microarchitecture. (1998) 69–77
9. Lee, C.C., Chen, I.C., Mudge, T.N.: The bi-mode branch predictor. In: Proc. Ann. ACM/IEEE Int. Symp. Microarchitecture, Research Triangle Park, NC (1997) 4–13
10. Calder, B., Grunwald, D., Zorn, B.: Quantifying behavioral differences between C and C++ programs. Journal of Programming Languages **2** (1994)
11. Yeh, T.Y., Patt, Y.: Two-level adaptive branch prediction. In: Proc. Ann. Int. Symp. Microarchitecture. (1991)
12. McFarling, S.: Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab (1993)
13. Sechrest, S., Lee, C.C., Mudge, T.: Correlation and aliasing in dynamic branch predictors. In: Proc. Ann. Int. Symp. Comput. Architecture. (1996) 22–32
14. Evers, M., Patel, S.J., Chappell, R.S., Patt, Y.N.: An analysis of correlation and predictability: What makes two-level branch predictors work. In: Proc. Ann. Int. Symp. Comput. Architecture, Barcelona, Spain (1998) 52–61
15. Thomas, R., Franklin, M., Wilkerson, C., Stark, J.: Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history. In: Ann. Int. Symp. Comput. Architecture, San Diego, CA (2003) 314–323
16. Yeh, T.Y., Patt, Y.: Alternative implementations of two-level adaptive branch prediction. In: Proc. Ann. Int. Symp. Comput. Architecture. (1992)
17. Nair, R.: Dynamic path-based branch correlation. In: Proc. Ann. Int. Symp. Microarchitecture. (1995) 15–23
18. Jacobson, Q., Rotenberg, E., Smith, J.E.: Path-based next trace prediction. In: Proc. Int. Symp. Microarchitecture. (1997) 14–23
19. Burger, D.C., Austin, T.M.: The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, WI (1997)
20. Shivakumar, P., Jouppi, N.P.: Cacti 3.0: An integrated cache timing, power, and area model. Technical Report (2001)
21. Stark, J., Evers, M., Patt, Y.N.: Variable length path branch prediction. In: Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems. (1998) 170–179

Power Efficient Instruction Caches for Embedded Systems

Dinesh C. Suresh, Walid A. Najjar, and Jun Yang

Department of Computer Science and Engineering, University of California,
Riverside, CA 92521, USA
{dinesh, najjar, junyang}@cs.ucr.edu

Abstract. Instruction caches typically consume 27% of the total power in modern high-end embedded systems. We propose a compiler-managed instruction store architecture (K-store) that places the computation intensive loops in a scratchpad like SRAM memory and allocates the remaining instructions to a regular instruction cache. At runtime, execution is switched dynamically between the instructions in the traditional instruction cache and the ones in the K-store, by inserting jump instructions. The necessary jump instructions add 0.038% on an average to the total dynamic instruction count. We compare the performance and energy consumption of our K-store with that of a conventional instruction cache of equal size. When used in lieu of a 8KB, 4-way associative instruction cache, K-store provides 32% reduction in energy and 7% reduction in execution time. Unlike loop caches, K-store maps the frequent code in a reserved address space and hence, it can switch between the kernel memory and the instruction cache without any noticeable performance penalty.

1 Introduction

Low power is a very important design criterion in the design of a very large number of embedded computing systems. Caches consume over 50% of the total energy of an embedded system [11]. The energy consumed by the instruction cache is of particular importance since an instruction is fetched every cycle. While numerous low-power instruction cache designs have been proposed in the literature, recent trends in research [3][7][9] have been directed towards customizing caches for embedded system applications. Examples of such customized instruction cache architecture include loop-cache like architectures [3][4] that place frequently executed loops on a special, smaller sized instruction cache.

It is a well-known observation that a software program spends 90% of its execution time in executing 10% of the code: a feature known as the 90–10 rule. The 90/10 (or 80/20) rule is even more relevant in embedded applications than desktop ones. In one of our previous works [17], we identified and quantified the execution kernels in a large number of embedded programs. The execution kernel is defined as a set of *functions and/or loops* that together account for a substantial percentage of the overall execution time. We found that the execution kernels often possessed a high execution density (execution count per unit size). Table 1 summarizes the kernel sizes for applications

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 182–191, 2005.

Table 1. Kernel and Program sizes, in bytes, and Static and Dynamic contributions of the Kernel for applications in the MediaBench and NetBench suites

Code	Kernel	Program	Kernel	Kernel
	Size (B)	Size (B)	% static	% dynamic
DRR	740	22511	03.28	45.12
Jpegencode	1996	102975	01.93	54.91
url	1688	11271	14.90	58.29
Unepic	2440	29727	08.20	59.40
Dh	1001	54563	01.83	66.51
Md5	7124	12895	55.24	66.57
G721enc	2470	250439	00.99	67.81
G721dec	2296	250439	00.91	68.67
Mpegencode	1576	96263	01.63	68.75
Tl	2336	20223	11.55	70.70
Mpegdecode	704	68035	01.03	79.12
Crc	584	7483	07.80	87.22
Adpcmencode	916	8091	11.32	96.17
Adpcmencode	1216	8192	14.84	97.12

from the MediaBench [8] and NetBench [10] benchmark suites. From the data reported the vast majority of kernels were less than 4KB in size and most of these were less than 2KB which is well within the size of a scratchpad memory as is commonly available in embedded processors [5]. Table 1 also shows the percentage contribution of the kernel to the total program size (static) and to the execution time (dynamic).

Let us consider the frequently executed code for the Diffie-Hellman Key exchange (DH) application [10] shown in Table 1. The kernel for this application constitutes 66.51% of the total dynamic instruction count and it contains a most frequent function (NN_DigitMult) that takes up nearly 35% of the total execution time. Hence, any loop-cache like architecture executing such applications must cache both loops and function calls. Accommodating such functions in loop-cache like architecture often increases the size requirement of a loop cache. Scratchpad memories take up much lesser area and consume nearly 40% lesser power than instruction caches of equal size [2] and consequently, they are ideal alternatives to loop caches.

In this paper we propose an instruction store architecture that is designed to exploit specific features of execution kernels in embedded applications. We call it the Kernel Store or K-Store. The main idea in the K-store is that a scratchpad like memory is used to store the execution kernel (*both loops and functions*) of the application: This kernel memory is therefore a fast SRAM memory at the level of the cache that can be accessed in one cycle. Unlike the cache it does not have to support tag arrays. The remainder of the application is stored in main memory and is accessed in the traditional way via an instruction cache. The system software is modified to support the kernel memory by inserting jumps where appropriate in the code. The compiler maps the kernel instructions to a separate region in the address space and hence, facilitates easy detection of these instructions during run-time.

Traditional loop cache architectures [3][4][9] cannot store frequently executed function calls inside the loop cache. K-Store overcomes this limitation by caching the kernel code (both functions and loops) in a tagless scratchpad memory. A 8KB direct-mapped basic K-store provides 28% reduction in energy while a 8KB, direct mapped supplemental K-Store provides 32% reduction in energy when used in lieu of a conventional instruction cache of equal size.

The rest of this paper is organized as follows. Section 2 illustrates the design of our K-Store architecture. In section 3, we describe our experimental framework. In Section 4, we discuss the results and evaluate the energy and performance of K-Store architecture. We provide a list of related research work in section 5. In section 6, we present concluding remarks.

2 K-Store Design

The K-Store architecture consists of two components: the kernel memory and the instruction cache. Figure 1 shows the block diagram of our K-Store architecture: The K-Store consists of both the kernel memory and the instruction cache. During run-time, instruction block requests are intercepted by a logic circuit, which identifies whether the requested address belongs to the kernel space or the non-kernel space.

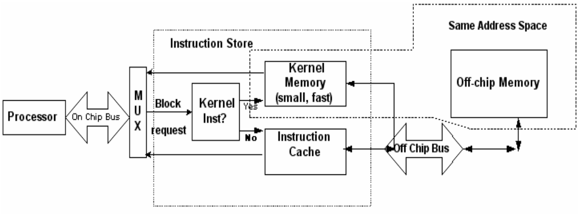


Fig. 1. K-store Architecture

By examining just a few bits of the instruction address during program execution, we can determine whether the instruction address lies in the kernel space or in the non-kernel space. For example, let us assume that for a given application, the compiler has mapped the kernel code in the address range of 4096 to 8192 bytes. We need a circuit to identify these kernel instructions at runtime. Figure 2 illustrates the operation of a logic circuit used to identify kernel instructions in this case. Here, the bit values in bit positions 1–12 are not useful in identifying the kernel instructions (don't care). When the 13^{th} LSB is high and all the remaining bits from the 14^{th} to the Most Significant Bit (MSB) are zeros, we can conclude that the instruction is a kernel instruction. A low value in all of the bit positions from 14^{th} to the MSB can be easily detected through the use of a five-levels of 2-input nor gate. While using 0.18-micron process technology, the access time of an 8k, 4-way set associative cache with block size of 32 bytes, as obtained using the CACTI tool [18], is 1.28 ns. A five level gate delay in 0.18-micron process technology typically amounts to 0.3 ns [14]. Hence, for a 500 MHz system (cycle time = 2 ns), the kernel detection logic can be easily accommodated within the same cycle. As shown earlier in Table 1, most of the kernel sizes are less than 4k and hence, checking for a high value on the 13^{th} or 14^{th} bit is sufficient to accommodate most of the application kernels. The K-store architecture is ideally suited for systems without a virtual memory.

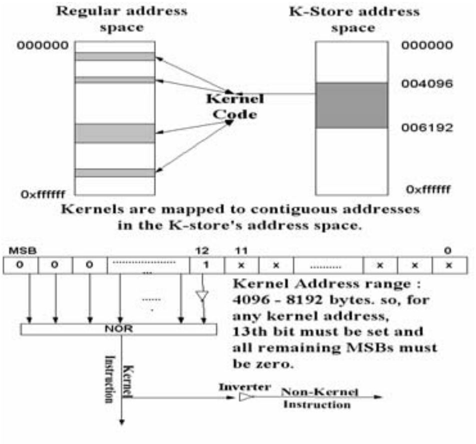

Regular address space
K-Store address space
000000
000000
004096
Kernel Code
006192
Oxffffff
Oxffffff
Kernels are mapped to contiguous addresses in the K-store's address space.
MSB
12
11
0
0 0 0 1 x x x x x
NOR
Kernel Address range : 4096 - 8192 bytes. so, for any kernel address, 13th bit must be set and all remaining MSBs must be zero.
Kernel Instruction
Inverter
Non-Kernel Instruction

Fig. 2. Runtime identification of kernel instructions

As shown in Fig. 2, the compiler maps the kernel instructions to a reserved area in the off-chip address space. Occurrences of the kernel code in the original program can be replaced by jump instructions that transfer the control flow to the kernel address space. Upon completing the execution of the kernel code, we need a jump instruction to continue program execution in the non-kernel address space. Thus every call to the kernel code adds two additional control transfer instructions to the program. Using our simulator, we measured the total increase in the number of control transfer instructions in the program. As shown in Table 2, we found that the control transfer instructions increased the dynamic instruction count by 0.039% on an average.

In order to better understand the cache behavior of the non-kernel portions of a program, we investigated the use of kernel memory with varying cache configurations. To ensure a fair comparison, we restricted the size of the k-store's instruction cache so that the total size of the kernel memory and the k-store's I-cache was equal to that

Table 2. Additional jump instructions to map kernel instructions of different programs

Benchmark	Additional dynamic control Instructions	Dynamic Instruction Count	% of additional Instructions
Adpcmencode	148	31481991	0.00047
G721decode	1595527	1005741879	0.158642
G721encode	1907982	1068726694	0.1785
Jpegencode	20455	81017999	0.02547
Mpegdecode	578160	1020616339	0.056648
Mpegencode	2344064	7037415745	0.033309
Unepic	236	30588223	0.000772
CRC	1256	18524458	0.00678
Dh	452608	12450027332	0.0003635
Drr	65	16266546	0.0004
MD5	59844	371031482	0.01614
Tl	597	2054152	0.029063
url	60330	1426337205	0.00423
Average	540097	1889217696	0.0392

of the baseline cache. By doing so, the K-store's instruction has a smaller size than the baseline cache and consequently, all accesses to the K-store are serviced at lesser power when compared to that of a baseline cache. We call these k-store cache configurations as basic K-store.

We also investigate the use of the kernel memory as a supplement to a baseline cache. Hence, we picked a baseline cache and added a small kernel memory to it and observed the energy reduction in this case. In spite of the higher cost associated with this design, the number of off-chip accesses would be much lesser than that of the baseline cache and hence, this design should be highly energy-efficient. For the rest of this paper, we will refer to this K-store configuration as supplemental K-store. We will discuss our experimental setup in the following section.

3 Experimental Framework

We analyzed an extensive collection of embedded system benchmarks from the NetBench [10](CRC, MD5, DH, DRR, TL and URL) and the MediaBench [8] (ADPCM, JPEG, MPEG and G721) benchmark suites. Table 3 gives a brief description about the benchmarks used in our experiments. For each of these applications, we used our loop analysis software [17] to identify the time consuming loops and function calls. We then identified the kernel instructions in these benchmarks and we extended the Sim-cache simulator supplied with the Simplescalar tool set [15] in order to simulate our design

We calculate the energy savings using the following formula:

$$\mathbf{Energy}_{K-Store} = \mathbf{Energy}_{Kernel} + \mathbf{Energy}_{Cache} + \mathbf{Energy}_{Off-Chip}$$

The total number of accesses to each of these components are obtained from the *s*im-cache simulator [15] we obtained the energy per kernel memory access and energy per cache access from the CACTI tool [18]. We investigate the use of K-store in two different configurations – basic K-store and the supplemental K-store. For our instruction store design, we vary the kernel memory size (2K, 4K), instruction cache size (2K, 4K), associativity (Direct, 4-way), off-chip miss penalty (20, 40, 60, 100, 200 cycles) and we evaluate the impact of these parameters on the energy savings and memory access latency. For supplemental K-store, we use a kernel memory of size 1KB and compare our K-store design with a conventional instruction cache. For each of these configurations, we fixed the cache block size at 32 bytes. We used 0.18-micron process technology in our power model. In the following section, we present the energy and performance results for our K-Store architecture.

Table 3. Average normalized energy and memory cycle reduction for different cache configurations

Cache Configuration	Energy Reduction	Cycles Reduction
4K Direct mapped	21%	3.6%
4K-4way set assoc.	20%	3%
8K-Direct mapped	28%	0.5%
8K-4way set assoc.	25%	0.1%

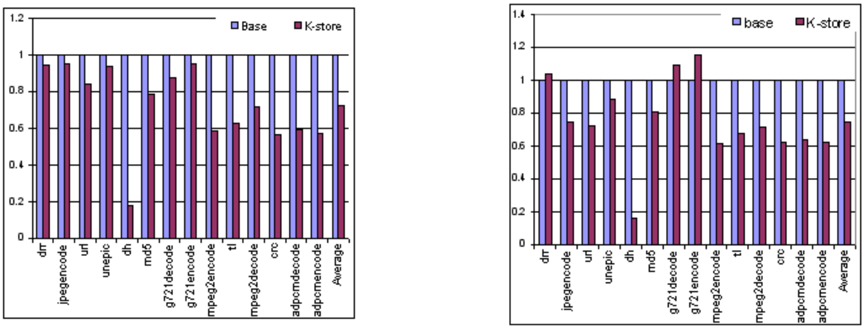


Fig. 3. Normalized energy consumption for (a) a direct mapped cache and (b) a 4-way set associative cache; Baseline= 8k, K-store: Kernel = 4k, Cache = 4k

4 Energy and Performance Evaluation

Basic K-Store. Figure 3a shows the normalized energy consumption for a 8Kb, direct mapped cache. The K-store uses kernel and cache memories of size 4Kb each. We find that K-Store provides an average energy reduction of 28% over a conventional instruction cache. Figure 3b shows the normalized energy consumption for a 8Kb, 4-way set associative cache. The K-store uses kernel memories and cache memories of size 4Kb each. In these graphs, we find that for the *Diffie-Hellman* key exchange application (*dh*), the energy savings is much higher than the rest. This is due to the fact that the *dh* kernel, has an extremely small static size and still contributes towards 66% of the total execution time. Applications like *adpcm* and *crc* are also characterized by very small static size and high execution count when compared to other applications under consideration. Hence, they yield significant energy savings.

Figure 4 shows the normalized cycle time for a 8Kb, direct mapped cache. The K-store uses kernel and cache memories of size 4Kb each. In spite of providing high-energy savings, the cycle time reduction for the *Adpcm* application is not so significant when compared to other applications. *Adpcm's* code size is comparable to the base cache size (8KB) and hence, the base cache provides a higher hit rate than the K-store's instruction cache (4KB). On an average, basic K-store yields a 0.5% reduction in the overall execution cycles.

We explored the design space to find out the optimal sizes of kernel memory and instruction caches for our instruction store architecture. In Table 3, we show the average normalized values of energy and memory cycle reduction for each of the cache configurations. For the results shown in Figures 3–6 and in Table 3, we assumed that an off-chip access was 60 times more expensive than an on-chip access. For each of the cache configurations, we also computed the values of energy savings and execution time reduction for varying values of off-chip penalties. Figure 5 provides the execution time reduction and the average energy savings for varying values of off-chip penalties.

The energy savings in the kernel memory are obtained due to two reasons: the smaller size of the kernel memory and the simple addressing scheme. Since the kernel memory is tagless, the tag comparison power, which contributes to 35% of the cache

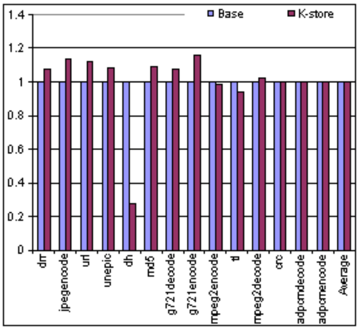


Fig. 4. Normalized execution time for a direct- mapped cache; Baseline = 8K, K-store: Kernel memory = 4K, cache = 4K

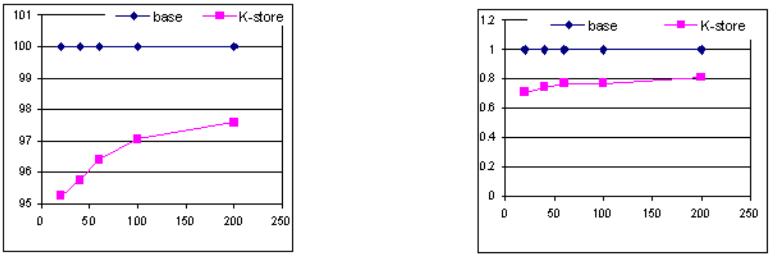


Fig. 5. Percentage reduction in (a) memory cycles and (b) energy for varying values of off-chip penalty

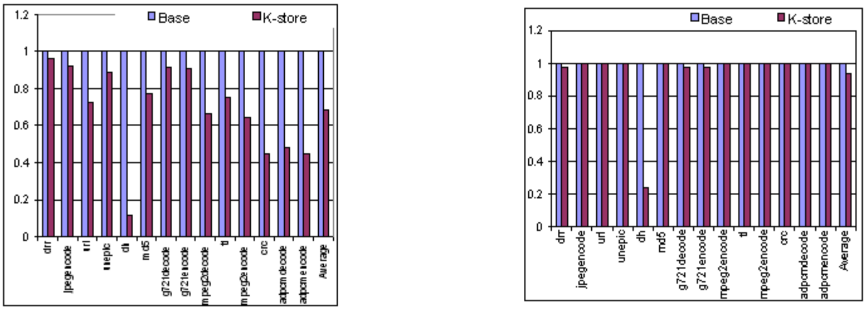


Fig. 6. Normalized (a) energy consumption and (b) execution cycles for a direct mapped cache; Baseline = 8K, K-store : Kernel memory = 1K, cache = 8K

power, is no longer necessary for a substantial fraction of the executed instructions. A smaller sized cache services the non-kernel instructions and hence, reduction in cache size is one of the main reasons for the energy savings reported in this paper. On an average, we found that reduction in cache size alone, accounted for 13% of the energy savings.

Supplemental K-Store. We also added the kernel memory to the best baseline cache configuration and evaluated the resultant energy and performance benefits. We found that when a 8KB, direct mapped instruction cache was augmented with a 1KB kernel memory, we achieved 32% reduction in energy and 7% reduction in execution time. The results are shown in Fig. 6. Since the instruction cache size is the same in the baseline cache and the K-store, K-store has fewer misses and hence, results in higher energy savings.

5 Related Work

Several techniques have been proposed to reduce the energy dissipation in the instruction caches of embedded processors. Many of these methods involve the usage of a tiny cache as a supplement to an existing instruction cache [9][2][7][13][19]. The tiny caches are designed in such a way that they exploit some feature of the application in order to capture most of the processor requests. By making sure that these smaller caches service bulk of the access request, significant amount of energy can be saved on each cache access.

Banakar et al. [2] have reported that a scratch-pad memory takes up 34% lesser area, consumes 40% less power and lowers cycle time by 18% when used in lieu of a cache of equal size. The power, cost and performance advantages provided by scratch-pad memories make them ideal candidates for replacing conventional caches. Researchers have rigorously investigated the use of a scratch pad memory to hold frequently used data items. Panda [12], Kandemir [6], and Avissar [1] have explored the use of scratch pad memory as a supplement for traditional cache architectures. They placed a small, fast, memory at the level of L1 cache and they use this memory to hold the most frequently used data items. Thus the off-chip traffic due to misses is reduced.

Avissar et al. [1] proposed an automatic compiler management strategy for data allocation amongst heterogeneous memory units. Panda [12] minimized the cross interference between different variables in the data cache by mapping them onto scratch pad memory and DRAMS. Sjodin [16] proposed a method wherein the critical variables with large number of accesses are stored on an on-chip SRAM while less critical variables allocated to a slower external RAM. Kandemir et al [6] proposed a compiler-directed on-chip software management strategy for data accesses. They store the reusable data values in nested loops onto an on-chip SRAM, thereby minimizing the data transfer between the off-chip memory and the on-chip scratch pad memory. K-Store uses the scratch pad memory to hold the frequently executed instructions and is hence, orthogonal to the works mentioned above, which focus on data storage.

Bellas et al. [4] have proposed the use of an L0 cache that resides between the CPU and the L1-cache. The compiler selects a few basic blocks to be placed in the L0-cache. Statically loaded loop caches (SLLC) [2] exploit compile-time information to preload the caches with the instructions of one frequently executed loop and hence, reduce the cold start misses.

K-Store is different from L0-cache [4] and SLLC [3] in the following aspects. Even though the SLLC and L0-cache exploit compile-time information, they can only extract simple tight loops that contain no function calls. This is due to the limitation of their access mechanism – by testing if the current PC falls within the range of the beginning and the ending addresses of the loops, the control logic decides whether the current instruction is a loop instruction stored in the SLLC or the L-cache. Such a mechanism excludes those loops containing function calls. While in the K-Store architecture, there is no such restriction since the entire loop body is moved to a different memory address space, simplifying the detection of the kernel instructions and increasing the number of candidate kernels. Besides, preloaded loop caches are architecturally more complex than scratch-pad memories.

Cache Aware Scratchpad Allocation (CASA) [20] provides a sophisticated technique for analyzing conflicts within the instruction cache and reduces these conflicts by using the scratchpad. By using the scratchpad to store both kernel blocks and conflicting instruction blocks, CASA can be effectively combined with a K-store to achieve significant energy benefits.

In the wake of aforementioned discussion, our contributions in this paper can be summarized as follows: We propose that program segments with high execution density (*both loops and functions*) should be held in a scratch pad memory (Kernel memory) and the remaining instructions can be efficiently cached in a regular instruction cache. In order to facilitate easy and non-intrusive detection of kernel instructions, we map the kernel instructions to a separate region in the off-chip address space. We illustrate that our approach is highly energy efficient.

6 Conclusion

Most of the embedded system applications tend to have strong kernels, which are instruction blocks with high execution count and low static size. In this paper, we propose a compiler-managed instruction store architecture that exploits kernel features to provide energy and performance benefits. Our compiler-assisted instruction store places the computationally intensive kernel code (*functions and loops*) onto a small, fast *scratch-pad memory* (*kernel memory*) and allocates the remaining instruction blocks to a regular instruction cache. 8Kb direct mapped supplemental K-store provides 32% reduction in energy and 7% reduction in execution time when used in lieu of a direct mapped cache of equal size.

References

1. Avissar, O., Barua, R., Stewart, D.: An Optimal Allocation for Scratch-Pad Based Embedded Systems. ACM Trans. on Embedded Computing Systems **1** (2002) 6–26
2. Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., Marwedel, P.: Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In: Proceedings of the 10th Int. Workshop on Hardware/Software Codesign, Estes Park, CO (2002)
3. Cotterell, S., Vahid, F.: Tuning of Loop Cache Architectures to Programs in Embedded Systems Design. In: IEEE/ACM Int. Symp. on System Synthesis (2002) 8–13
4. Bellas, N., Hajj, I., Polychronopoulos, C., Stamoulis, G.: Energy and Performance Improvements in Microprocessor Design Using a Loop Cache. In: Int. Conf. on Computer Design (1999) 378–383

5. Intel Corp. Intel XScale (tm) Core Developer's Manual, 2002. http://developer.intel.com/design/intelxscale/.
6. Kandemir, M., Kadayif, I., Sezer, U.: Exploiting Scratch-Pad Memory Using Presburger Formulas. In: Int. Symp. on System Synthesis, Montreal, Canada (2001) 7-12
7. Kin, J., M. Gupta, M., Mangione-Smith, W.H.: The Filter Cache: An Energy Efficient Memory Architecture. In: the 30th Annual IEEE/ACM Symp. on Micro Architecture
8. Lee, C., Potkonjak, M., Smith, W.H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In: Int. Symp. on Microarchitecture Research Triangle Park, NC (1997) 292–303
9. Lee, L., Moyer, B., Arends, J.: Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with small tight loops. In: Int. Symp. on Low Power Design (1999)
10. Memik, G., Smith, W.H., Hu, W.: NetBench: A Benchmarking suite for Network processors. In: Proc. of Int. Conf. on Computer-Aided Design (ICCAD), San Jose, CA (2001) 39–42
11. Montanaro, J., et al.: A 160MHz, 32b, 0.5W CMOS RISC Microprocessor. IEEE Journal of Solid State Circuits (1996) 1703–1714
12. Panda, P.R., N.D. Dutt, N.D., Nicolau, A.: Efficient Utilization of Scratch-Pad Memory in Embedded Processor applications. In: Proc. of European Design and Test Conf., Paris (1997)
13. Ravindran, R., Nagarkar, P.D., Dashika, G.S., Marsman, E.D., Senger, R.M., Mahlke, S.A., Brown, R.: Compiler Managed Dynamic Instruction Placement in a Low-Power Code Cache. In: Proc. of the 3rd Intl. Symp. on Code Generation and Optimization (CGO) (2005)
14. http://www.semicon.toshiba.co.jp/eng/prd/asic/topix.html
15. Simplescalar Simulator. http://www.simplescalar.com
16. Sjodin, J., Von Platen, C.: Storage Allocation for Embedded Processors. In: International Conference on Compiler, Architecture and Synthesis for Embedded Systems (CASES 2001), Atlanta, GA (2001)
17. Suresh, D.C., Najjar, W.A., Vahid, F., Villarreal, J., Stitt, G.: Profiling Tools for Hardware/Software Partitioning of Embedded Systems. In: Proc. of ACM SIGPLAN conference of Language Compilers and Tools for Embedded Systems (LCTES), San Diego, CA (2003) 189–198
18. Steven, J. Wilton, E., Jouppi, N.P.: CACTI: An Enhanced Cache Access and Cycle Time Model. IEEE Journal of Solid State Circuits, **31** (1996) 677–688
19. Tang, W., Gupta, R., Nicolau, A.: Power Savings in Embedded Processors through Decode Filter Cache. In: Proceedings of the Design Automation and Test in Europe (2002)
20. Verma, M., Wehmeyer, L., Marwedel, P.: Cache-Aware Scratchpad Allocation Algorithm. In: Design Automation and Test in Europe (DATE), Paris, France (2004)

Micro-architecture Performance Estimation by Formula

Lucanus J. Simonson[1] and Lei He[2]

[1] Intel Corporation, Santa Clara CA 95052, USA
[2] University of California, Los Angeles CA 90095, USA

Abstract. An analytical performance model for out of order issue superscalar micro-processors is presented. This model quantifies the performance impacts of micro-architecture design options including memory hierarchy, branch prediction, issue width and changes in pipeline depth at all pipeline stages. The model requires a minimal number of cycle accurate and trace driven simulations to calibrate and once calibrated estimates performance by formula. The model estimates the performance of arbitrary micro-architecture configurations with an average error of 6.4%. During early design stages when cycle accurate simulation is prohibitive an analytical model can provide guidance to designers to increase design quality and reduce design effort. This allows the design of an embedded processor to be rapidly tuned to its application by reducing the cost of exploring the design space.

1 Introduction

During early planning stages of micro-processor design a clear understanding of the impact various design decisions will have on performance is critical. Cycle accurate simulation is often used to collect performance information, but is too time consuming to be well suited to exploration of a large number of competing design options and does not lead directly to an understanding of why a change in the micro-architecture impacts performance. Because performing a large number of cycle accurate simulations is prohibitively a faster method of producing performance information is needed. We present a super-scalar, out of order issue microprocessor performance model that estimates performance by formula and requires a minimal number of cycle accurate and trace driven simulations to calibrate. This model allows greater freedom to explore micro-architecture options and pipeline strategies during early design, and provides the capability to more easily tune a processor to its application.

Our model is capable of estimating performance for different combinations of micro-architecture options and pipeline depth at all stages of instruction execution. Micro-architecture choices include issue width, resource contention, memory hierarchy, branch predictor strategy and instruction prefetch. Combining these design elements into a single performance model enables designers and design tools to optimize all of these parameters simultaneously with minimal simulation time. The model is intuitive and reasonably accurate. It requires constant runtime once built up and provides insight into the causes of performance loss and the ways in which they interact in a realistic processor. This work also establishes the methodology for generating a similar model for

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 192–201, 2005.

other base micro-architectures such as those used in embedded processors or under development in industry. Up front simulation costs are minimized by decoupling all of the various factors that contribute to performance loss and recombining them analytically. Empirical cycle-accurate sensitivity analysis of pipeline depth and trace driven simulation of the behavior of each micro-architecture design variable in isolation are used to provide the inputs to the model.

Related to this paper, many approaches have been proposed to estimate performance. A theoretical method for analyzing in-order pipelines is presented in [1], however the work does not apply to out-of-order issue architectures. Both in-order and out-of-order front-end pipeline depth is analyzed in [2], though the backend was not. While exploring the impact of increasing pipeline depth on processor performance in the Pentium processor [3], an empirical performance model was developed to estimate performance as a function of pipeline depth, but no micro-architecture changes were considered.

A first-order superscalar processor model [4] provides a formula based estimate of performance. It assumes an idealized micro-architecture free of resource contention and capable of sustained throughput equal to issue width and estimates the performance penalties due to branch mispredictions and instruction and data cache misses from a trace-driven simulation. Building upon [4], we consider a more realistic micro-architecture that includes resource contention and instruction prefetch. We also decouple the behavior of the various caches and branch predictor so that a trace-driven simulation is not required for each unique combination of cache and branch predictor settings. In addition, we consider the performance impact of varying the backend pipeline depth and combine that with the performance impact of the frontend analytically by employing a novel probabilistic performance loss event overlap model.

The rest of the paper is organized as follows. In Section 2 we present background information on architecture design options considered, simulation engine used and our methodology for deriving the model. In Section 3 we present our analytical performance model. We experimentally verify the correctness and accuracy of our model in Section 4 and conclude the paper in Section 5.

2 Background

2.1 Micro-architecture Design Space

The Micro-Architecture considered in this work is a super-scalar out of order issue CPU that includes the seven modules shown in Table 1 with the options considered for each. Pipeline depth is simulated by clock cycle latency between the micro-architecture modules defined in Table 1. These interconnects are listed in Table 2. The symbol used to represent the latency of each interconnect is listed as well as a qualitative description of the type of performance degradation caused by latency on each.

2.2 Methodology

All simulations were performed using modified version of SimpleScalar 2.0 with PISA instruction set architecture in truncated runs with a fastforward period of forty million instructions and a sample period of twenty million instructions. Six benchmarks

Table 1. Micro-architecture Design Freedoms and Options

Design Freedom	Options
Issue Width	2, 4, 8
Integer ALU Number	Equal to issue width, 3/4 of issue width
Other Arithmetic Unit Number	1/4 Integer ALU, 1/2 Integer ALU

Branch Predictor Size/Strategy	bimod 1K, BTB 128; bimod 2K, BTB 256;
	combined bimod 2K, 2-level 1K, BTB 512;
	combined bimod 4K, 2-level 2K, BTB 1K

Instruction Level 1 Cache	8KB Direct Mapped, 16KB Direct Mapped, 32KB 2 Way Associative, 64KB 4 Way Associative
Data Level 1 Cache	8KB Direct Mapped, 16KB Direct Mapped, 32KB 2 Way Associative, 64KB 4 Way Associative
Unified L2 Cache	128KB 2 Way Associative, 256KB 4 Way Associative, 512KB 4 Way Associative, 1MB 8 Way Associative

Table 2. Interconnect Pipeline Design Freedoms

Symbol	Interconnect	Performance Impact
$L_{IL1/L2}$	IL1 Cache to L2 Cache	Increased IL1 Cache Miss Penalty
$L_{DL1/L2}$	DL1 Cache to L2 Cache	Increased DL1 Cache Miss Penalty
L_{fetch}	Fetch Unit to IL1 Cache	Increased Branch Misprediction Penalty Prefetch Penalty
$L_{dispatch}$	Fetch Unit to Dispatch Unit	Increased Branch Misprediction Penalty
L_{issue}	Dispatch Unit to Issue Unit	Increased Branch Misprediction Penalty
L_{DL1}	Issue Unit to DL1 Cache	Stalls on data load dependencies
L_{IALU}	Issue Unit to each I-ALU	Stalls on integer dependencies Increased Branch Misprediction Penalty
L_{IMult}	Issue Unit to each I-Multiplier	Stalls on multiply dependencies
L_{FALU}	Issue Unit to each FP-ALU	Stalls on floating point dependencies
L_{FMult}	Issue Unit to each FP-Multiplier	Stalls on floating point dependencies

(mcf, equake, art, mesa, parser and bzip2) from the SPEC2000 suite were evaluated to produce all experimental results presented. These include a mix of floating point and integer benchmarks and were chosen to represent a range of real world application behaviors. Performance of an architecture is summarized as the arithmetic mean of CPI for the six benchmarks.

The model was developed by performing a study of the performance impact of each design variable in isolation while holding all other design variables constant. The performance impact of the design variable was graphed and, if possible, a mathematical formula derived to fit the observed behavior. The formulas and empirical constants obtained by studying each design variable in isolation are combined incrementally, grouping variables by category, developing mathematical expressions for the interaction between the behavior of each variable with other variables in its category and finally between categories. This led to the division into front and backend in the model and the grouping of terms.

Deriving the expressions to describe the interaction between design variables in terms of their performance impact relied upon insight into the anatomy of a performance loss event. From [4] we know that some performance loss events are not inter-related. We verified these findings and proceeded to the new factors considered by our model. For each interaction we devised a hypothetical expression based upon insight into the micro-architecture. We tested each hypothesis by performing an experiment to isolate the interaction and measure by cycle accurate simulation.

3 Analytic CPI Model

We measure performance in terms of cycles per instruction (CPI) which we define as the average number of clock cycles per instruction issued by the processor on the correct execution path. CPI is proportional to execution time and is convenient for our purposes because the delay caused by some performance loss event directly adds to execution time, can simply be averaged over the number of instructions and added to the CPI. Our approach to performance modeling is to count the number of performance loss events and quantify their delay penalties. Should two delay penalties be incurred at the same time the performance overhead should not be double counted. We correct for overlapping performance loss events to obtain an accurate performance estimation formula.

3.1 Model with Interconnect Pipeline

The CPI of a micro-architecture is a combination of performance loss due to miss events, data dependency stalls and the baseline performance of the micro-architecture with no extra pipelining in the absence of miss events. Baseline performance is a function of the issue width and resource constraints of the micro architecture combined with the amount of instruction level parallelism in the benchmark. The miss events we consider are branch mispredictions, level one instruction cache misses and level two instruction and data cache misses. Level one data cache misses are modeled the same as the latencies of arithmetic units that result in data dependency stalls.

We quantify the performance impact of latency in each of the interconnects from Table 2 in terms of contribution to miss penalty and average duration of data dependency stall, then combine their performance impacts to estimate system performance with consideration of pipelined interconnect. Arithmetic units of the same type are grouped together into a single module and have identical interconnect latency. We divide the interconnects into frontend and backend interconnects. Frontend interconnects are those that contribute latency to the pipeline prior to the issue stage. The equation for the micro-architecture CPI model with consideration of pipelined interconnect is given in (1).

$$CPI = CPI_{ideal} + CPI_{IL1} + CPI_{L2} + CPI_{Front} + CPI_{Back} \tag{1}$$

3.2 Cache CPI Overhead

Miss rates for one type of level one cache are clearly independent of the other level one cache configuration as well as the level two cache configuration. Given that the level two cache is sufficiently larger than the level one caches, our experiments show its miss rates are roughly independent of level one cache configuration. For this reason the miss rates for each cache option are measured independent of the configuration options chosen for the other caches reducing the number of trace driven simulations required to build up the model.

In equation (1) performance loos due to instruction level one cache misses, CPI_{IL1}, is defined as the access latency of the L2 cache, $L_{L2(access)}$, plus the interconnect latency between the IL1 and L2 cache, $L_{IL1/L2}$, multiplied by the miss rate of the instruction level one cache. Instruction cache miss penalty is equal to the latency of the next higher level of memory hierarchy [4].

$$CPI_{IL1} = MissRate_{IL1}(L_{L2(access)} + L_{IL1/L2}) \tag{2}$$

CPI_{L2} is the performance loss due to level two cache misses, broken down into instruction and data cache miss rates, multiplied by the latency to main memory, L_{MM}. The $F_{OverlapL2(data)}$ term in (3) is the expected value for the size of a group of level two data cache misses that all occur within the issue window size number of instructions of the previous level two data cache miss. This factor accounts for the overlapping of L2 data cache miss performance loss as described in [4]. We assume the miss penalty for level two cache to be the latency of a main memory access. Unlike [4] we do not calculate or use an overlap factor between level two data cache misses and other miss events because we assume that fetch is blocked by the time the L2 data miss penalty is incurred, preventing such overlap from occurring.

$$CPI_{L2} = (MissRate_{L2(inst)} + \frac{MissRate_{L2(data)}}{F_{overlapL2(data)}})L_{MM} \tag{3}$$

3.3 Frontend CPI Overhead

Two sources of performance loss contribute to frontend CPI, branch misprediction and prefetch overhead. When latency is added between the level one instruction cache and the fetch logic the branch predictor does not have the opportunity to decide whether a branch is taken until several clock cycles after it has been read from the cache. Prefetch proceeds to fetch contiguous blocks in memory until a branch predicted as taken reaches the fetch unit. The prefetch pipeline must then be flushed and fetching resumes at the target address. The formula for CPI_{Front} is given in (4).

$$CPI_{Front} = \alpha(CPI_{BPred(pipe)} + CPI_{Prefetch}) + CPI_{BPred} \tag{4}$$

CPI_{BPred} is the performance loss due to branch misprediction

$$CPI_{BPred} = MissRate_{BPred}Penalty_{Intrinsic} \tag{5}$$

where the intrinsic branch misprediction penalty, $Penalty_{Intrinsic}$, is measured for a given benchmark by cycle accurate simulation for a single micro-architecture by dividing the difference between CPI_{ideal} with and without perfect branch prediction by the branch misprediction rate, $MissRate_{BPred}$. In our experiments we observed that the branch misprediction penalty had minimal dependence upon the branch prediction option chosen or the issue width of the microprocessor and is instead a function of pipeline depth and performance loss overlap. For this reason we use a single measurement of intrinsic, baseline branch misprediction penalty for all micro-architecture configurations. The branch misprediction rate is measured by trace driven simulation of each of the branch predictor options in Table 1. The $CPI_{BPred(pipe)}$ term in (4) is defined as

$$CPI_{BPred(pipe)} = MissRate_{BPred}Penalty_{BPred(pipe)} \tag{6}$$

where $Penalty_{BPred(pipe)}$ is defined as

$$Penalty_{Bpred(pipe)} = 2(L_{fetch} + L_{dispatch} + L_{issue} + L_{IALU}) \tag{7}$$

such that pipeline stages added anywhere in the integer pipeline contribute two cycles of branch misprediction penalty. The justification for this is that when a branch is being issued the instruction it depends upon (which we assume to be an integer instruction) has often not yet committed, so adding one cycle of latency anywhere in the integer pipeline will add one cycle of branch misprediction penalty due to data dependency delay of branch issue. If there is no data dependency then adding one cycle of latency to the frontend will add one cycle of branch misprediction penalty due to delayed resolution time of the branch. In both cases a cycle of latency in the backend will add one cycle of branch misprediction penalty due to delayed resolution time of the branch. Finally, adding one cycle of latency specifically to the frontend pipeline will add an additional cycle of branch misprediction penalty because the frontend pipeline takes longer to refill after it has been flushed due to a branch misprediction.

The $CPI_{Prefetch}$ quantity in (4) is defined as

$$CPI_{Prefetch} = \frac{HitRate_{BPred}Branches_{Taken}L_{fetch}}{Instructions} \tag{8}$$

where the ratio of $Branches_{Taken}$ to $Instructions$ is the probability that a given instruction is a taken branch. Multiplying by the branch prediction hit rate, $HitRate_{BPred}$, factors out the prefetch flushes that overlap performance loss due to branch mispredictions for those same branches.

3.4 Frontend Overlap Correction

Because performance loss in the frontend may overlap other performance loss events we introduce in (4) a correction factor, α, based upon the approximation that the timing of all potentially overlapping performance loss events are independent and random. The equation for this correction factor is

$$\alpha = \frac{CPI_{unit}}{CPI_{ideal} + CPI_{Bpred} + CPI_{BPred(pipe)} + CPI_{back}} \tag{9}$$

where CPI_{unit} is similar to CPI_{ideal} but with an initial latency of one on all arithmetic operations including the typically long latency floating point divide and root operations. The CPI_{unit} quantity can be thought of as the time the processor spends performing useful work. The α overlap factor is the probability that performance loss events due to

Table 3. Fitness of α overlap factor

IL1 Fetch Latency	1	2	4	8
Error	2.4%	1.7%	4.0%	11.6%

added frontend pipeline stages do not overlap some other source of performance loss, which is the probability that they happen during the time that the processor is performing useful work. Branch misprediction can overlap other branch misprediction events as well as data dependency stalls. It cannot overlap prefetch performance loss or instruction cache miss performance loss because during these periods no branches can be fetched. When fetch becomes blocked due to extremely long latency data dependency stalls due to level two data cache misses no overlap can occur.

To demonstrate the fitness of modeling the probability of overlap between performance loss events as independent random variables with the α overlap factor we measure cycle accurate CPI for our set of benchmarks while letting the interconnect latency, L_{fetch} vary from one to eight while other micro-architecture freedoms are held constant and compare to cycle accurate simulation. As presented in Table 3 the error for this experiment is very low at small latencies and only 11.6% at extreme fetch latency.

3.5 Backend CPI Overhead

Performance loss for the interconnects in the back end is due to data dependency stalls. The stall incurred by a dependent instruction is determined by the maximum commit time of the instructions it is dependent upon. It does not matter which of the backend interconnects contribute latency to the maximum commit time because the effect is the same. Backend interconnects are the same way and can be modeled as a group by summing their individual models.

$$CPI_{Back} = \sum_{x\varepsilon\{IALU,IMult,FALU,FMult,DL1\}} CPI_x \tag{10}$$

Because performance loss in the back end is linear CPI overhead of individual interconnects can be accurately modelled as in (11) where C_x is an empirical constant obtained in a similar way to $Penalty_{BPred}$ by calculating the slope of a line between two cycle accurate CPI measurements while varying L_x.

$$CPI_x = C_x L_x \tag{11}$$

In the case of data loads the latency is increased by the latency to go to the level two cache with probability equal to the level one data cache miss rate.

$$CPI_{DL1} = C_{DL1}(L_{DL1} + MissRate_{DL1}(L_{DL1/L2} + L_{L2(access)})) \tag{12}$$

We verify our backend pipeline model by checking it against cycle accurate simulation. In this experiment we use the baseline architecture used to calculate CPI_{ideal} but let the latencies to the arithmetic units vary uniformly by factors of two from one to sixteen. Maximum estimation error for this experiment was around 2% showing that the backend model based upon linear performance penalty as a function of latency is highly accurate.

4 Experiments

We verify the correctness and accuracy of our performance model in an experiment where thirty-two random configurations chosen from the options in Table 1 and latencies ranging from zero to nine inclusive from the interconnects in Table 2 are measured by cycle accurate simulation.

The cycle accurate CPI for each of the thirty two configurations is plotted as the independent variable while the model estimate is plotted as the dependent variable. As we can see the data points line up nicely along the $y = x$ line. The model shows good fidelity, with average error for this experiment of 6.4% and maximum error of 18.4%. We compare this to the error reported in [4] of 4.4% on average and 12% in the worst case in which performance is estimated for a subset of five out of the sixteen design variables we consider here. We succeed in decoupling the behavior of the various caches and the pipeline depth as well as adding in consideration of pipeline depth in the back end without paying too high a price in accuracy over the state of the art.

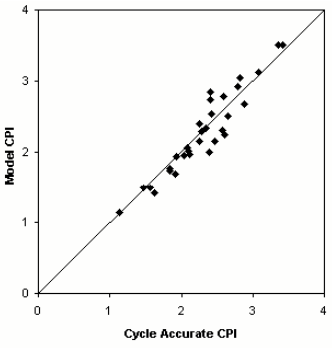


Fig. 1. Lumped Arithmetic Module Model Fidelity

Table 4. Average absolute error of individual benchmarks

art	bzip2	equake	mcf	mesa	parser	average
7.3%	10.1%	8.8%	8.8%	7.2%	9.5%	8.6%

In Table 4 the average absolute error is broken down by benchmark. One of the goals of the model is to abstract away the contribution of instruction stream to performance. Based upon the assumption that this was possible, cycle accurate simulation of only a single benchmark, equake, from the set of six mentioned in section 2.2 was used to derive the form of the model as discussed in section 2.2. This was necessary during the initial micro-architecture study because of the large number of cycle accurate simulations required. By comparing the accuracy of the full model against other benchmarks in Table 4 we validate the ability of the model to abstract away the contribution of instruction stream to performance.

The average absolute error reported for Fig. 1 differs from that in Table 4 because it is calculated by summing the CPI of the benchmarks before calculating absolute error, in effect treating the six benchmarks sampled as a single benchmark six times as long. Significantly, the difference is small because the model tends to underestimate CPI, as can be observed by comparing the number of data points below and above the $y = x$ line in Fig. 1. This tendency is a contributing factor to the model's good fidelity.

5 Conclusions and Discussions

We have developed an accurate analytical performance model for superscalar out of order issue microprocessors. It models changes in cache hierarchy, branch predictor size and strategy, issue width and pipeline depth at all stages of execution. The model provides performance estimates accurate to within 6.4% on average and 18.4% in the worst case for a random set of micro-architecture design options and pipeline depths. This compares to 4.4% on average and 12% in the worst case as reported in [4], which serves as the basis for our work but considers only a third of the micro-architecture design freedoms in our model. Additionally our model provides further insight into what the causes of performance loss are in modern micro-processors and how these sources of performance loss interact with each other to determine the overall performance of a micro-processor.

The model requires that two cycle accurate simulations be performed for each issue width under consideration, an additional cycle accurate simulation to obtain the intrinsic branch misprediction penalty of a benchmark and one cycle accurate simulation for each type of arithmetic unit employed by the micro-architecture. The model also requires a single trace driven simulation for each cache configuration option and each branch predictor configuration option under consideration. The relatively low cost of building up the model is the key feature that makes it practical. The entire range of possible combinations of micro-architecture options and pipeline depths can be explored with a minimal amount of simulation. This provides the basis for a speed/accuracy trade-off giving designers the option to choose between the analytical model and cycle accurate simulation to suite their needs.

Modular design of a processor targeting a specific application could benefit from rapid exploration of the micro-architecture design space, considering all of the discrete options in cache size, branch prediction, ALU type and issue width to minimize area while meeting performance constraints. This type of model provides the capability to perform rapid "what if" analysis of micro-architecture design choices, enabling designers to immediately see the impact a change can be expected to have on system performance or enabling automated design tools to execute interactive refinement optimization algorithms such as floorplanning with consideration of performance.

Future work includes the following objectives: developing an analytical power model based upon the analytical performance model, applying the analytical performance model to iterative automated micro-architecture and floorplanning co-optimization such as in [5], modeling the interaction between performance loss events in micro-architectures utilizing different branch misprediction recovery mechanisms, extending the analytical performance model to multi-core processors and eliminating the need for cycle accurate simulation in building up the analytical performance model by replacing empirical constants with formulas based upon instruction stream statistics.

References

1. Emma, P.G., Davidson, E.S.: Characterization of branch and data dependencies on programs for evalutating pipeline performance. IEEE Trans. on Computers **36** (1987) 859–875
2. Hartstein, A., Puzak, T.R.: The optimum pipeline depth for a microprocessor. In: International Symposium on Computer Architecture. (2002)
3. Sprangle, E., Carmean, D.: Increasing processor performance by implementing deeper pipelines. In: International Symposium on Computer Architecture. (2002)
4. Karkhanis, T., Smith, J.E.: A first-order superscalar processor model. In: International Symposium on Computer Architecture. (2004)
5. C. Long, C., Simonson, L., Liao, W., He, L.: Floorplanning optimization with trajectory piecewise-linear model for ipelined interconnects. In: DAC. (2004)

Offline Phase Analysis and Optimization for Multi-configuration Processors

Frederik Vandeputte, Lieven Eeckhout, and Koen De Bosschere

Ghent University, Electronics and Information Systems Department,
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{fgvdeput, leeckhou, kdb}@elis.UGent.be

Abstract. Energy consumption has become a major issue for modern microprocessors. In previous work, several techniques were presented to reduce the overall energy consumption by dynamically adapting various hardware structures. Most approaches however lack the ability to deal efficiently with the huge amount of possible hardware configurations in case of multiple adaptive structures. In this paper, we present a framework that is able to deal with this huge configuration space problem. We first identify phases through profiling and determine the optimal hardware configuration per phase using an efficient offline search algorithm. During program execution, we inspect the phase behavior and adapt the hardware on a per-phase basis. This paper also proposes a new phase classification scheme as well as a phase correspondence metric to quantify the phase similarity between different runs of a program. Using SPEC2000 benchmarks, we show that our adaptive processing framework achieves an energy reduction of 40% on average with an average performance degradation of only 2%.

1 Introduction

Energy dissipation is a major design issue for modern microprocessors both in the embedded, the general-purpose as well as the high performance market segments. To address this issue several researchers have proposed to dynamically tune or resize several hardware resources without affecting overall performance, thereby reducing energy consumption.

Generally speaking, we can identify three major classes of adaptive processing: resource-driven, positional and temporal adaptation. In resource-driven adaptation [1][2], the various hardware components tune themselves according to their current use. In positional adaptation [3], particular architectural configurations are associated with particular code sections, for example at the level of subroutines. Each time the processor enters one of those sections, the corresponding architectural configuration is installed. In the temporal approach [4][5][6], the program execution is partitioned into fixed-length intervals, recurring phases are identified and energy-efficient processor configurations are associated per phase. Phase identification and hardware adaptation is all done dynamically. A common problem is how to deal efficiently with a large number of hardware configurations in case of multiple adaptive structures. Finding the optimal configuration through enumeration (as is typically done in previously proposed dynamic optimization schemes) obviously is not an option.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 202–211, 2005.

This paper makes the following contributions. First, we present an efficient offline phase analysis framework to reduce the overall energy consumption on multi-configuration processors. Second, we propose a new phase classification method that combines clustering efficiency with phase predictability. Third, we propose a phase correspondence metric to quantify the phase similarity between different runs of a given program with different inputs. The overall end result is an offline phase-based method that reduces the energy consumption by 40% while incurring a performance degradation of only 2%.

In the next section we will discuss some previous work. Section 3 details our experimental setup. In section 4, we present our adaptive processing framework. Overall results are presented in section 5. Finally, we conclude in section 6.

2 Previous Work

As already mentioned, there exist many hardware adaptation techniques to reduce the energy consumption of a microprocessor [7][4][1][5][3][6]. Nearly all previously proposed approaches however lack the ability to deal efficiently with a large processor configuration space when many hardware structures can be resized. This is due to the fact that determining the optimal hardware configuration per phase is typically done through online enumeration. This paper proposes a practical solution to this problem by determining the phases and the optimal hardware configuration per phase offline. An additional advantage of doing the phase analysis offline is that it provides a global view on the complete program execution. As such, intervals can be classified by taking a global view of the program phase behavior into account. Dynamic approaches on the other hand need to take greedy choices during program execution. We will now highlight two approaches that closely resemble the approach taken in this paper.

In [6], a purely dynamic temporal phase detection and prediction scheme is presented from which the basic framework in this paper is extracted: the way phase information is collected using footprints and the fact that the next phase ID is predicted. However, this does not affect the generality of our work, i.e. the offline configuration space exploration could also be used in conjunction with other ways of collecting phase information such as instruction working set signatures [4], etc., or in case there is no phase prediction available.

In [3], three different implementations of positional adaptation schemes are proposed, of which SISD (static instrumentation and static decision) resembles our approach most. With SISD, all major subroutines are first identified by profiling the program. Then a number of Low Power Techniques (LPTs) are evaluated on each of those subroutines during a number of profiling runs. At the end, an efficiency score is computed for each subroutine-LPT pair and for each of those subroutines the most efficient configuration is chosen. Subsequently, the program needs to be recompiled to trigger the appropriate hardware adaptation at entry and exit points of the selected subroutines. A major disadvantage of the SISD approach is that the program must be profiled multiple times. In case of n LPTs, $n+1$ profiling runs are needed: n with each LPT enabled and one with no LPTs enabled. In addition, each profiling run requires a complete benchmark execution which can be very costly.

3 Methodology

Before presenting our framework we first detail our experimental setup. We use cycle-level processor simulation using Simplescalar/Alpha v3.0 [8] in combination with Wattch v1.02 [9] to collect performance and energy consumption data. For Wattch, we assume a 0.18 μm-technology and a 1.2GHz clock frequency. Our baseline processor is depicted in Table 1. We use a subset of the SPEC2000 benchmark suite, see Table 2. We used these benchmarks since they exhibit a sufficiently diverse phase behavior. The binaries were taken from the SimpleScalar website. These programs were all simulated from start to completion using the reference inputs given in Table 2. We used the train inputs for our offline phase analysis and per-phase optimal hardware configuration exploration.

Table 1. Baseline simulation model

Processor Width	8-wide fetch/decode/issue/commit - double fetch speed
Functional Units	8 int ALU - 4 mem ports - 2 FP ALU - 2 int MULT/DIV - 2 FP MULT/DIV
Buffers	128 entry re-order buffer - 32 entry load/store queue
Branch Predictor	hybrid: 13-bit 8k 2-level predictor - 8k bimodal predictor - 8k meta predictor
L1 I-cache	8KB 2-way set-associative - 32 byte blocks - 2 cycle latency
L1 D-cache	16KB 4-way set-associative - 32 byte blocks - 2 cycle latency
L2 unified cache	1024KB 4-way set-associative - 64 byte blocks - 20 cycle latency
Main Memory	151 cycle latency

Table 2. The SPEC2000 benchmarks, their ref. input and the number of phases

benchmark	ref input	# phases	benchmark	ref input	# phases
eon	rushmeier	1	ammp	ref	11
gcc	200.i	30	applu	ref	32
gzip	graphic	18	equake	ref	10
mcf	ref	9	facerec	ref	27
twolf	ref	5	swim	ref	25
vpr	route	5	wupwise	ref	8

4 Offline Phase Capturing and Architecture Tuning

In this section, we will discuss the various components of our adaptation scheme in detail: collecting footprints, phase classification, configuration space exploration, adaptive execution and phase prediction.

4.1 Collecting Footprints

In [10], Sherwood *et al.* propose the use of the Basic Block Vector (BBV) to summarize program behavior of a fixed length instruction interval in a hardware-independent way. Each element of a BBV contains the number of times a basic block gets executed during that interval, multiplied by the number of instructions in that basic block. In order to be able to efficiently capture program phase behavior in hardware, Sherwood *et al.* [6] came up with the notion of a footprint which is a condensed form of a BBV. A footprint is computed by maintaining an accumulator table of 32 24-bit saturating counters. The accumulator table is updated for each executed branch. At the end of the fixed-length interval (1M instructions in this study), the 8 most significant bits from each accumulator entry are extracted to form a 32-byte footprint.

4.2 Phase Classification

Once the footprints are collected for a given program, we can classify them into phases. One particular static phase classification approach is employed in SimPoint. In SimPoint [10], offline clustering is done through *k*-means clustering [11].

Because the *k*-means algorithm does not say anything about the optimal number of clusters, SimPoint performs the clustering algorithm for a number of values of *k* (ranging from 1 to K_{max}) and computes the Bayesian Information Criterion score (BIC) [11] to quantify the clustering quality. The optimal *k* is then determined by choosing the clustering having a BIC score that is at least 90% of the spread between the best and the worst BIC score over a range of different *k*'s.

In our approach we also use *k*-means clustering but we use a different metric than BIC to guide phase classification by weighting clustering quality versus phase predictability. The idea behind our proposal is that on the one hand we want phases that exhibit similar behavior within the phases and dissimilar behavior between the phases—this can be achieved by having a large number of phases. On the other hand, we want to be able to predict future phases highly accurately so that we do not lose energy reduction opportunities nor experience performance degradation due to incorrect future phase predictions. As having less phases could result into a number of actually different phases to be clumbed together, we also incorporate a special transition phase (as also proposed in [12]), grouping all intervals that do not belong to any of the *k* phases. As intervals belonging to that transition phase will not be optimized, it is important to keep the number of intervals assigned to this transition phase small.

Since clustering quality generally increases and phase predictability generally decreases with an increasing number of clusters, there exists an optimal clustering in which clustering quality and phase predictability are weighted appropriately. To this end, we propose the Quality-Predictability Score (QPS) which is defined as the multiplication of the clustering quality (CQ) metric and phase predictability. The CQ metric is defined as one minus the average distance of each point to its cluster center relative to the maximum distance D_{max} of a point to the center of the entire data set[1]. The optimal clustering is then defined as the one with the maximum QPS.

Figure 1 shows an example for gzip in which the CQ metric, the phase predictability, the QPS and the BIC score are shown. The optimal clustering in this example is obtained for *k* around 20, but as one can see, there are a number of other clusterings that have almost the same QPS-score. Applying the BIC scoring mechanism would result in 120 to 130 phases, which results in a phase prediction accuracy of only 57%. This means that the adaptive processor would be in the wrong phase in almost half of the time.

One potential weakness of offline analysis is that it relies on profiling using a train input. In our case, the phases as observed when running the train input should resemble

[1] To penalize the amount of intervals belonging to the transition phase, these intervals all have distance D_{max}, thus decreasing the clustering quality.

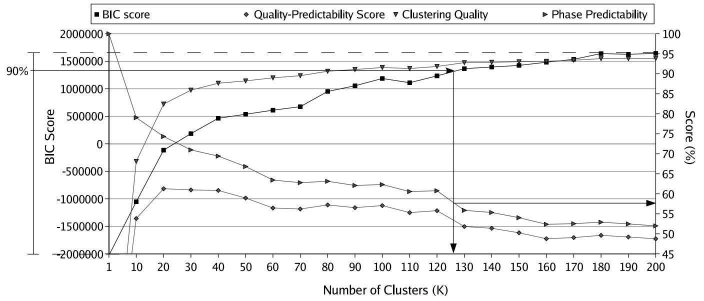


Fig. 1. The quality-predictability score (QPS) combining clustering quality (CQ) with phase predictability. An example for gzip

Table 3. The PCM for the various benchmarks

benchmark	PCM	benchmark	PCM
eon	94.44%	ammp	94.98%
gcc	70.83%	applu	82.82%
gzip	85.67%	equake	91.10%
mcf	78.70%	facerec	98.65%
twolf	83.35%	swim	96.19%
vpr	94.29%	wupwise	99.29%

the phases as observed when running the reference input. Ideally, there should be a one-to-one mapping of phases. In order to validate the quality of the profiling information we have done the following experiment. We have calculated the *train phases* as well as the *reference phases* when running the train and reference input, respectively. Subsequently, we have computed the *phase correspondence matrix* between these phases. An element (i, j) in the phase correspondence matrix gives the number of fixed length intervals (when running the reference input) corresponding to reference phase i and train phase j. Based on the correspondence matrix we now calculate the *phase correspondence metric* (PCM) which quantifies how well the phase behavior of the train and reference input correspond to each other. The PCM is calculated as follows. Every entry in the matrix is first weighted by the similarity between the corresponding train and reference phase. The similarity between a train and a reference phase is computed as one minus the (normalized) distance between the footprints of the centroid of the train and reference phase respectively. All those weighted entries in the matrix are summed and this sum is then divided by the total number of intervals when running the reference input. Obviously, if there is a one-to-one correspondence in phase behavior between the train input and the reference input, the PCM will be one. As such, the closer to one the PCM is, the better. Table 3 shows the PCM for the various benchmarks. We observe that there is a fairly good correspondence between the train input and the reference input, with an average phase correspondence of 89%. One exception is gcc which has a phase correspondence of only 70%. One could try however to combine phase information from multiple inputs in order to increase the quality of the profiling information. This is subject of future work.

4.3 Configuration Space Exploration

Now that the phases are available, we can determine the optimal architectural configuration per phase. This is done through an offline configuration space exploration. We use detailed simulation for this purpose on a representative interval per phase. Again, being able to determine the representative interval offline provides the possibility to carefully choose a representative interval using a sophisticated algorithm. Online configuration space exploration on the other hand, has to take greedy decisions again; typically, the first few intervals from a phase are considered to determine the optimal hardware configuration. In our offline analysis, we choose the interval being closest to the cluster center as the representative interval for the given phase.

In our configuration space exploration, we determine the hardware configuration that attains the maximum energy reduction while achieving an IPC that is within 2% of the IPC of the baseline configuration. We vary the branch predictor size, the processor width, the number of functional units, the window and fetch buffer sizes and the number of active ways in the caches. Most of these resizable structures are controlled by clock gating and sleep transistors. For the buffers, special care must be taken to shrink them appropriately. If the size to shrink to is smaller than the current number of occupied entries, resizing is postponed until the number of occupied entries drops below the target size; the latter is accelerated by disallowing new entries to enter the buffer until resizing is possible. For the cache adaptation, we employ the 'active cache ways' approach as proposed by Albonesi [7]. Under this cache adaptation strategy, a subset of the cache ways can be disabled to reduce energy consumption; in fact, only the data array is disabled, the tag array remains untouched. When disabled, a cache way still preserves its contents. When an access is done to an inactive way, a penalty of four cycles is incurred and the data is transfered to one of the active ways.

Given the large search space (we have 1.89×10^{15} possible configurations in this study), it is impossible to do an exhaustive search. As such, we propose some simplified but effective variants of the steepest descent (SD) search heuristic called *static steepest descent* (SSD) and *adaptive static steepest descent* (ASSD). Steepest descent tries to find the optimal solution in a given search space by following the path with the largest slope for a given optimization function. The main problem of steepest descent however is that it requires many evaluations in each step if the dimensionality of the search space is high (e.g. in our case, we vary 21 hardware parameters during each step to know the highest slope in that point).

A simple greedy solution to drastically reduce the number of evaluations is to evaluate the effect of reducing each parameter only in the first step, order all parameters according to their slope in that initial point, and then optimize each parameter in that order one after another while keeping the rest of the configuration untouched. This is what we call static steepest descent (SSD) because we only evaluate the slope of each parameter once, and use the same ordering in the following steps. The main disadvantage of this algorithm is that the next parameter is only optimized if the previous parameter has been optimized. In many cases however, it is better to only partially optimize a parameter so that the next parameter can be optimized more aggressively. Indeed, a parameter that was initially *cheap* to optimize can become very costly after a while in such a way that other parameters now have become cheaper. However, we do not want to evaluate all parameters in each step. Instead, we build a table with the slope in the initial design point as is done is SSD. During the search algorithm we do the following in each iteration: we determine the architectural parameter with the largest slope and optimize along that parameter. We then update the table with this newly obtained slope. If the slope for the current parameter is smaller than one of the other slopes in the table, we undo the last optimization step and continue the optimization process with the parameter in the table that now has the largest slope. This optimization algorithm is called adaptive static steepest descent (ASSD) and combines the advantages of both SD and SSD.

In order to further reduce the total time spent in the configuration space exploration we add a simple but effective pruning scheme to the search algorithms. Recall that we start from our baseline configuration which has all its parameters at its maximum values. In all of our search algorithms we optimize each parameter by scaling down the corresponding hardware structure. So, if the IPC is no longer within 2% of the baseline IPC, we can stop optimizing the current parameter—scaling further down will not yield a higher IPC. We thus prune a part of the search space. Likewise, if the energy reduction starts decreasing, we also do not consider the current parameter anymore in the next steps.

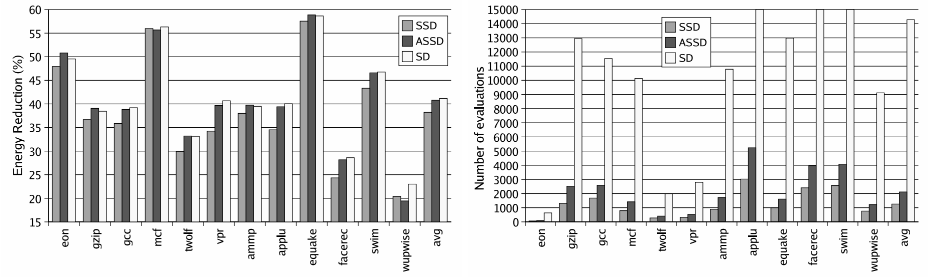


Fig. 2. The overall energy reduction (left) and total number of evaluations (right) for the search algorithms when optimizing all phases of each benchmark

Looking at Fig. 2, we conclude that ASSD performs nearly as well as SD in terms of energy reduction, but requires only a small number of evaluations. The average number of evaluations is about 132 for optimizing one phase and 2000 for one benchmark. SSD also performs reasonably well compared to ASSD and SD in terms of energy reduction, and the number of evaluations during design space exploration are yet smaller than for ASSD. SD requires a very large number of evaluation runs. In all cases presented in Fig. 2, the performance degradation was no more that 2%. To validate the effectiveness of these algorithms, we also evaluated these algorithms in a more limited search (to be able to compare to exhaustive searching) and concluded that the optimum identified through these simple heuristics was found to be close to the global optimum. For example, the energy reduction results of ASSD are within 5% of the optimal results on average. We did not include this evaluation in this paper due to lack of space.

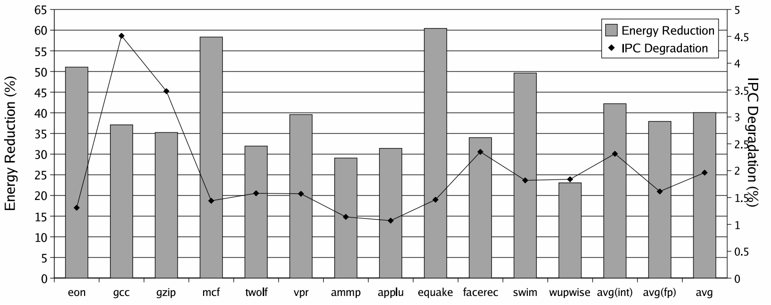


Fig. 3. Overall energy reduction and IPC degradation for our adaptive processor compared to the baseline processor

4.4 Adaptive Execution

From these offline analyses, we obtain a phase list consisting of a representative footprint (the footprint of the representative interval) and an optimal hardware configuration per phase. Upon program execution, the phase list is communicated to the hardware and is stored in the *Phase Information Table (PIT)*. The PIT contains a small number of entries (32 in our study, large enough to capture all phases of a program in this paper), each entry containing a phase ID, a 32-byte footprint (the footprint of the representative interval) and the optimal hardware configuration. During execution, phase information of the current interval is collected similar to what is explained in section 4.1. By the end of the interval, a footprint is obtained. This footprint is then compared to all the footprints in the PIT and the phase ID of the least distant footprint is returned. The phase history information and phase predictor are updated with the last phase ID and is subsequently used to index the phase predictor. The phase predictor then returns the next phase ID. Accessing the PIT with the predicted next phase ID yields the optimal hardware configuration for the next phase.

4.5 Phase Prediction

An important aspect of our temporal hardware adaptation approach is phase prediction. Phase prediction is used both for classifying intervals into phases as well as for predicting the next phase during program execution. In our framework, we used a 2-level burst predictor with conditional update and confidence, as Vandeputte et al. [13] showed that this is today's most accurate phase predictor. In this paper, we used a 256 entry 4-way set associative predictor table with 2-bit saturating counters and a confidence threshold of 1. The average phase misprediction rate of the SPEC2000 benchmarks when using the offline phases during the adaptive execution is 7.7% (compared to 14.5% when using the simple last value predictor).

5 Overall Results

Figure 3 shows the overall energy reduction and IPC degradation using our adaptation framework. Note that these results are obtained through a cross validation setup. We

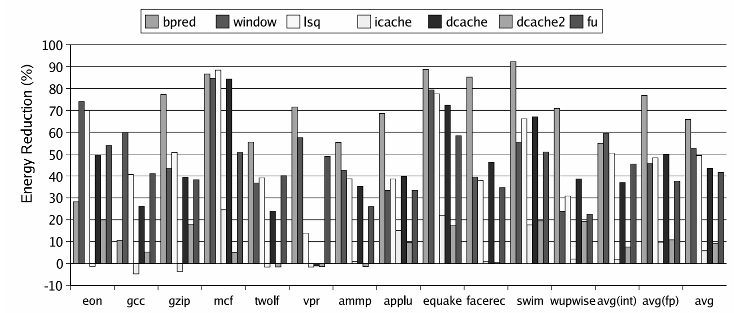


Fig. 4. Energy reduction per processor component

used the train input to identify the phases and to determine the optimal configuration per phase. The reference inputs are then used to report the energy reduction and performance degradation results. The results in Fig. 3 show an average energy reduction of 40% and a 2% average performance degradation. For most benchmarks, the performance degradations that we observe in Fig. 3 are around 1.5% to 2.5%. There are however a few exceptions, for example gcc and gzip, for which we observe a performance degradation of 4.5% and 3.5%, respectively. The reason for these relatively high IPC degradations is because of the mismatch between the phases of the program executed with the train input versus the reference input on the one hand, and the poor phase predictability (74% for gcc and 72% for gzip) on the other hand.

Figure 4 quantifies the energy reductions per processor component. We make a distinction between the branch predictor, the RUU, the LSQ, the L1 I-cache, the L1 D-cache, the L2 cache and the functional units. The largest energy reductions are observed in the instruction windows (both the RUU and the LSQ). The energy reductions in the branch predictor, the L1 D-cache and the functional units are also significant. For the L1 I-cache and the L2 cache on the other hand, we do not obtain that large energy reductions, 6% and 9% on average. There is clearly also a difference in energy reduction of various components between the SPECint and SPECfp programs. For example the branch predictor and L1 I-cache can be reduced more aggressively for SPECfp programs. The reason for this is that SPECfp programs have small, regular kernels, whereas SPECint programs are far more irregular.

6 Conclusions

Adaptative processing is an effective means for reducing the energy consumption without affecting overall performance. Although several approaches have been proposed for hardware adaptation, very few papers considered efficient managing of multi-configuration hardware. The biggest challenge here is how to deal with the large configuration space in case of multiple configurable units. In this paper we presented a framework that is able to efficiently deal with multi-configuration hardware. Next to this, we also (i) presented a highly efficient offline configuration space search algorithm that employs search space pruning and simulation of small instruction intervals, (ii) proposed a new phase classification metric and (iii) introduced a phase correspondence metric to quantify the phase similarity between different runs of a program. The overall energy reduction that we obtain is 40% on average with a 2% performance degradation.

Acknowledgements

This research was funded by Ghent University and by the Fund for Scientific Research-Flanders (FWO-Flanders).

References

1. Dropsho, S., et al.: Integrating adaptive on-chip storage structures for reduced dynamic power. In: Internat. Conf. on Parallel Arch. and Compil. Techniques. (2002)
2. Huang, M., et al.: Profile based energy reduction for high-performance processors. In: Workshop on Feedback-Directed and Dynamic Optimization. (2001)
3. Huang, M.C., Renau, J., Torrellas, J.: Positional adaptation of processors: application to energy reduction. In: Internat. symp. on Computer architecture. (2003)
4. Dhodapkar, A.S., Smith, J.E.: Managing multi-configuration hardware via dynamic working set analysis. In: Proc. of the Internat. symp. on Computer Arch. (2002)
5. Huang, M., et al.: A framework for dynamic energy efficiency and temperature management. In: Proc. of the Internat. symposium on Microarchitecture. (2000)
6. Sherwood, T., Sair, S., Calder, B.: Phase tracking and prediction. In: Proc. of the Internat. symposium on Computer architecture. (2003) 336–349
7. Albonesi, D.H.: Selective cache ways: on-demand cache resource allocation. In: Proc. of the 32nd Internat. symposium on Microarchitecture. (1999) 248–259
8. Burger, D., Austin, T.M.: The simplescalar tool set, version 2.0. SIGARCH Comput. Archit. News **25** (1997) 13–25
9. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: a framework for architectural-level power analysis and optimizations. In: Proc. of the Internat. symposium on Computer architecture. (2000) 83–94
10. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: Proc. of the 10th Int. Conf. Arch. Support Program. Languages Operating Syst. (2002) 45–57
11. Pelleg, D., Moore, A.: X-means: Extending k-means with efficient estimation of the number of clusters. In: Proc. of the Internat. Conf. on Machine Learning. (2000)
12. Lau, J., Schoenmackers, S., Calder, B.: Transition phase classification and prediction. In: Proc. of the Internat. Symposium on High Performance Computer Architecture. (2005)
13. Vandeputte, F., Eeckhout, L., De Bosschere, K.: A detailed study on phase predictors. Submitted to Europar 2005 (2005)

Hardware Cost Estimation for Application-Specific Processor Design

Teemu Pitkänen, Tommi Rantanen, Andrea Cilio, and Jarmo Takala

Tampere University of Technology, P.O. Box 553, FIN-33101 Tampere, Finland
{teemu.pitkanen, tommi.rantanen, andrea.cilio, jarmo.takala}@tut.fi

Abstract. In this paper, a methodology for estimating area, energy consumption and execution time of an application executed on a specified processor is proposed. In addition, a design exploration process to find suitable processor architectures for a specific application is proposed. Cost and performance estimation is an important part of the exploration process. The actual cost estimation is based on predefined characterizations of cost and performance of resources stored in a database. The results show that the method is quick and its accuracy is sufficient for design space exploration.

1 Introduction

In general, tailoring a processor architecture for a specific application or set of applications under certain implementation constraints calls for analyzing several architectural alternatives. This analysis requires several tasks to be performed: program code for the application has to be generated for each target architecture, performance of the code on each architecture has to be evaluated, and implementation costs have to be analyzed. A huge effort is required to perform all these tasks manually.

This problem can be alleviated with tool-assisted exploration of a vast architecture design space and a high-level language compiler that is retargetable at run time. In addition, the estimates of the cost of running an application on its target architecture, e.g., execution time, area, and energy, should be obtained. If hundreds of different architecture alternatives are to be analyzed, it is essential that the estimations can be obtained quickly.

In this paper, a methodology for estimating area, energy consumption, and execution time of an application executed on a processor is proposed. In addition, a design exploration process to find suitable processor architectures for a specific task is proposed. The actual cost estimation is based on predefined resource characteristics, which are stored into a database used by the exploration process. The estimates obtained with the proposed methodology are compared to reference values obtained from commercial simulation tools. The results show that the accuracy of our method is sufficient for the exploration process and the estimation is extremely quick compared to traditional methods.

2 Related Work

The work presented in this paper is based on the Move framework, a design environment developed at Delft University of Technology, Delft, the Netherlands. The frame-

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 212–221, 2005.

work consists of a set of tools for designing application-specific programmable processors [1]; it includes a retargetable high-level language (HLL) compiler, a cycle-accurate simulator and a processor generator. The framework provides also tool-assisted architecture exploration; the designer defines the maximum set of processor resources and the explorer estimates the cost of executing the application on different architectures. The processor architectures supported by the framework are all based on the same template. A target architecture is an instantiation of this template with a set of parameters. The tools of the framework can exploit the specific features of each instantiation.

The architecture template of the Move framework is based on transport triggering paradigm [2] —hence the name transport triggered architecture (TTA)— where an instruction specifies only the data transports to be performed by the interconnection network. The execution of an operation is a side effect of an operand transport to a specific operand register of a function unit. TTA's, therefore, remind the data-flow computation paradigm, but the availability of operands is determined statically.

In a TTA processor, only one type of operation is supported: the move operation, which performs a data transport from a source to destination. A TTA processor, as illustrated in Fig. 1, consists of a set of function units (FU) and register files (RF) containing general-purpose registers, a control unit, and an interconnection network consisting of buses. Data memory accesses are performed with the aid of a load-store unit, which is a normal function unit from the architecture point of view. The function units and register files can have different number of input and output ports, which are connected to one or more buses.

Each function unit begins to execute a new operation when an operand is moved into a trigger operand register, shown in the block diagram in Fig. 2. All the other operands have to be transferred into the operand registers in the same or earlier cycles. A function unit can produce one or more results. If the function unit supports several operations, the actual operation is selected by means of an opcode attached to the move that writes the trigger register. The function units can be pipelined and the latency of a unit is managed by the compiler, which expects deterministic latencies. Dynamic latency has to be managed at run-time by transport pipeline lock mechanism.

The design space exploration process used in Move framework, as described in [1], is an iterative process where, in each iteration, the target processor architecture is varied and the performance of the given application running on the processor is evaluated. The

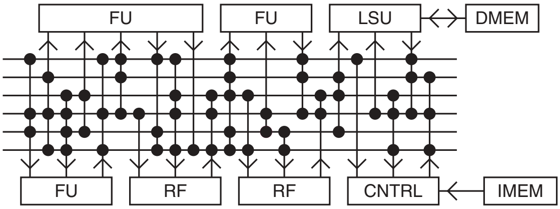


Fig. 1. TTA processor organization. FU: Function unit. RF: Register file. LSU: Load-store unit. CNTRL: Control unit. DMEM: Data memory. IMEM: Instruction memory. Dots represent connections between buses and ports of function units

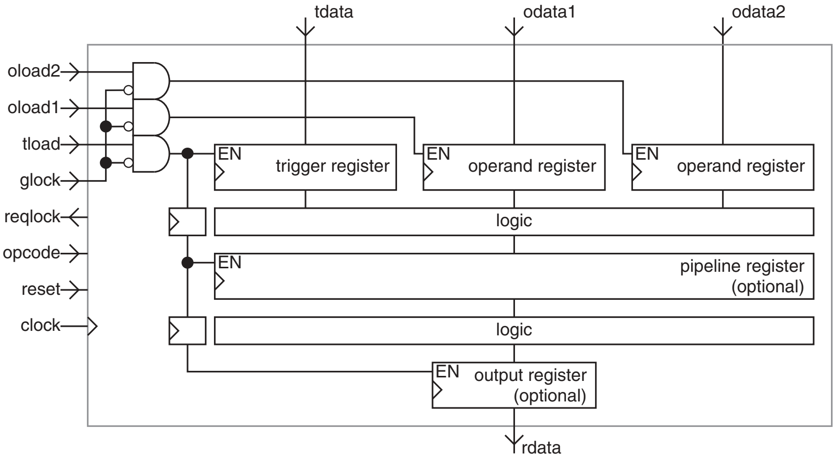


Fig. 2. Structure of a three-inputs, one-output function unit in a TTA processors

user defines the maximum set of resources, i.e., the number of buses, the number and type of FUs, etc. At each iteration of the exploration process, a target architecture, defined by a subset of the maximum resource set, is selected. The given application is mapped and scheduled onto the target architecture and simulated to obtain performance statistics. Finally, estimates of area and speed of the architecture on target technology are computed.

In the next iteration, one resource is removed and the map-schedule-simulate-estimate process repeated. Based on the obtained statistics, the next target architecture is selected by removing another resource. This process is repeated until a *critical* resource is removed, i.e., the application cannot be scheduled without the resource. The critical resource is put back and another one is removed. After a while, the target architecture contains only critical resources and nothing can be removed. At this point, the explorer begins to put resources back, but in different order. The explorer performs a predefined number of such remove-add resource sweeps. After the exploration is completed, the user can pick the target architectures with favorable cost and performance characteristics for further optimizations.

The design space exploration process requires estimation of various cost metrics. Different cost metrics of a digital circuit, e.g., area and power consumption, can be estimated at different levels of abstraction. In general, the estimation is a trade-off between accuracy and speed, i.e., better accuracy requires more details of the implementation. There are several tools to obtain accurate estimates of area, delay, and power consumption using physical models. These tools require that the VLSI implementation is available implying that the time consuming place and route phases have been completed. There are also low-level power analysis methods operating at RT-level, which provide comparable accuracy, e.g., [3] [4]. However, the low-level tools are not especially useful in architectural exploration due to their long simulation times. In [5], several methods of physical modeling, such as Rent's rule and Donath's wiring model, have been applied for evaluating the area and delay of the protocol processor architectures. The Rent's exponents were allocated for each hardware resource in the processor and final cost is obtained with linear approximation. The area accuracy is within 40% of true area.

Most of the efforts in power estimation have targeted at compiler optimizations, i.e., creating power-aware software. For this purpose, cycle-by-cycle power estimation is required. The estimation is often targeted to a specific microarchitecture, as with the methods reported in [6] [7]. If architectural alternatives are to be explored, the estimation method has to support higher level models. E.g., in [8], cycle-level performance simulator is used to obtain the hardware access counts. These counts are used to obtain power estimate with the aid of parameterizable power models of the resources. These models are based on effective capacitance and fall into four categories: array structures, fully associative content-addressable memories, combinational logic and wires, and clocking. Since the power models are based on capacitance, the models cannot be obtained automatically from a RT-level design. This implies manual work when new models are created.

In [9], the power estimation of a processor core is based on switching capacitance tables. Each function unit is analyzed with all the possible operand combinations. The power accuracy was verified within 10% of results obtained with circuit level simulators while the execution time was less than 0.1 sec per transition. The tabular power figures implies large tables when the number of bits in the inputs is increased.

The estimation models can also be obtained automatically from RT-level components as described in [10]. The elementary components are synthesized onto target technology and characteristics are stored into a component-library. The characteristics include both the delay and power consumption. The RT-level components are fairly fine-grained, e.g., full-adders and logic gates, thus complex systems will contain a large number of components. A power estimation method based on a higher level model is proposed in [11], where each functionality of a peripheral device is modeled as an instruction. Each instruction has a corresponding power mode, which allows a power-per-instruction look up table to be created. The power consumption of the processor is still based on measuring the current when a certain instruction is executed.

In the work described in this paper, the objective is architectural design space exploration, thus huge number of different processors are to be analyzed. This calls for an extremely quick estimation method. In addition, the absolute accuracy of the estimation results is not the main concern; the purpose is to obtain the relative cost of different design alternatives. After exploration, the user will pick up the most promising candidates for further investigation.

3 Cost Estimation Method

Due to the fact that the design space exploration is an iterative process and a large number of different architectures are analyzed, the cost estimation needs to be quick. In addition, the estimation method has to be independent from technology, i.e., the same tools should work when the target technology is changed. Finally, the exploration is used to pick up candidates for further optimizations, thus the relative cost is needed rather than the absolute cost.

In this work, we have exploited the fact that the exploration process contains already simulation, which provides statistics of the utilization of each resource. In addition, the framework contains a processor generator, which generates synthesizable hardware description of the final processor structure. This generator relies on library components, i.e., synthesizable hardware descriptions of resources such as function units, register files, etc. Thus, descriptions of most resources are predefined.

3.1 Modified Design Space Exploration Process

In the original exploration process in [1], an important architecture parameter was fixed before the exploration process; the latency of function units was given by the user. This approach does not take into account that the latency of a function unit is dependent on the timing requirement, i.e., a unit may require pipelining, which increases the latency, when shorter delay is needed. In a similar fashion, the same unit may have a different structure if delay requirements are different. E.g., simple ripple-carry adder is sufficient for low clock frequencies but higher clock frequencies may require carry-look ahead adders, thus the area and power of the units depends on the delay constraint.

In order to automate the selection of the FU latency, we propose that the cost information of several implementations of the same function unit is stored into a cost database together with latency and critical path information. Given a delay constraint, the latency of a function unit can be easily determined. This requires that the clock frequency is not estimated like in [1]. The cycle time is either given by the user or is varied by the exploration process.

The implementation of the function unit can be selected with the aid of the cycle time and delay of the component when the inputs and outputs of the function unit are registered, i.e., the critical path determining the delay is from a register to a register. However, the output register is optional, thus it is possible that the critical path in a complete processor is from the input register through the logic of the function unit and the interconnection to the input register of another function unit. In such a case, the interconnection delay has to be taken into account when determining whether the component fulfills the given timing constraints or not.

The proposed design space exploration process is illustrated in Fig. 3. The inputs of the exploration process are the unscheduled code of the given application, a set of processor components (resources), and the cost database. In each iteration, a target architecture is formed by selecting a set of resources. The architectural parameters that are related to the implementation, such as latency, are obtained from the cost database. The given application is then scheduled onto the target architecture and the parallel code is simulated. Simulation results are finally used to estimate the cost of executing the parallel code on the architecture. Each evaluated target architecture is stored with its cost and performance statistics as one design point in the design space.

3.2 Technology Characterization

Implementation-specific data about the resources is stored in the cost database. These data can be obtained by running logic synthesis tools and hardware simulators and analyzing the results. This process, called characterization, can be automated and performed off line, since the same cost database can be shared by several exploration runs, with different applications and initial architectures. The hardware description of each function unit is stored in libraries. Those descriptions are needed during the processor

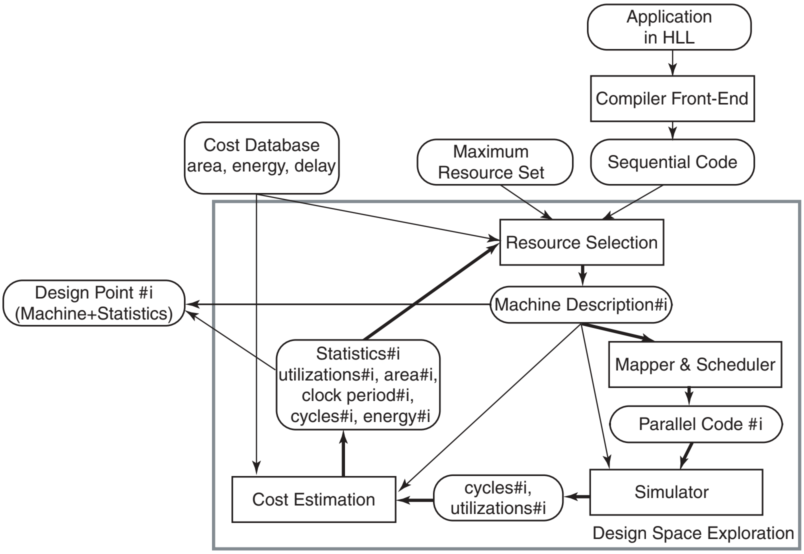


Fig. 3. Principal flow diagram of the proposed design space exploration process. HLL: High-level language

Table 1. Properties characterizing the hardware resources

Resource	Characterized by
Function Unit	operations, word width, operation delay (critical path), output delay (delay from last register to output), latency, no. pipeline stages, pipeline control discipline, no. input ports, no. output ports
Register File	no. words, word width, delay, no. read ports, no. write ports
Interconnect	fanin, fanout, word width, delay
Control Unit	density of the interconnection network

generation phase. During the characterization phase, a block is synthesized with varying synthesis constraints and simulated. The implementation data is obtained from the report files.

Each type of hardware resource, i.e., function units, register files, and interconnections, is characterized by a specific set of properties, as shown in Table 1. The database contains also an entry for the control logic. In addition, certain metrics may consist of several values for each entry.

3.3 Area Estimation

The total area of the target architecture is obtained as the sum of the area of each hardware resource, i.e., area of function units, register files, interconnection, and control unit. The area of a particular function unit and register file is obtained by querying the corresponding entry from the database. If an equal match does not exist in the database, the closest possible database entries are used for estimating the area.

The area estimation of interconnect is obtained with a slightly different approach. The interconnect consists of sockets, i.e., connections of a port to buses, as illustrated in Fig. 4. In our case, busses are realized with the aid of one-directional bit lines instead of tristate buses as in [1]. The input and output sockets contain AND gates and multiplexers, respectively. The actual bus contains an OR gate, thus the area estimate is obtained by summing the area estimates of those components from the cost database.

3.4 Energy Estimation

The estimate of the energy consumed by a resource cannot be obtained directly from the database but it must be linearly approximated from the utilization, and weighted according to the used cycle time. The energy estimate of a function unit, E^{FU}, is obtained as follows

$$E^{FU} = \left(\sum_i U_i E_i\right) + E_{idle}\left(n_c - \sum_i U_i\right) + E_{static}\frac{n_c t_{clk}}{t_d} \tag{1}$$

where E_i is the dynamic energy per clock of operation i, E_{idle} is dynamic energy when no operation is executed per clock, E_{static} is the energy due to leakage current during time t_d, t_d is the delay of critical path, U_i is the number of times the operation i is used, n_c is the number of cycles executed in the simulation, and t_{clk} is the clock period. The parameters E_i, E_{idle}, E_{static}, and t_d are obtained from the cost database, while the parameters U_i and n_c are obtained from the instruction set simulation. The clock period t_{clk} is defined during the resource selection.

The energy of a register file, E^{RF}, requires a different formulation. An RF is characterised by its number of input and output ports, which has a strong effect on the power consumption. The formula used to compute the energy is

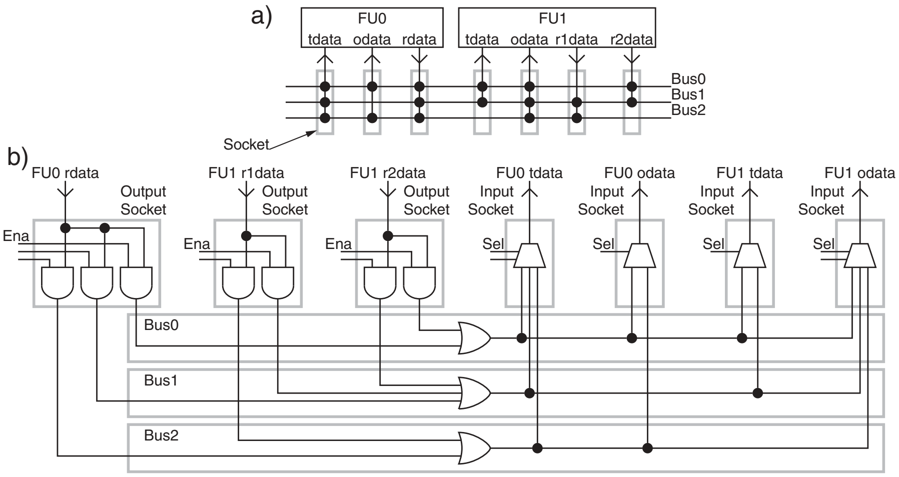


Fig. 4. Example of TTA processor: (a) logical diagram and (b) principal structure

$$E^{RF} = \left(\sum_{r} \sum_{w} E_{rw} U_{rw} \right) + E_{static} \frac{n_c t_{clk}}{t_d} \tag{2}$$

where E_{rw} is the dynamic energy per clock when r reads and w writes are performed in parallel, U_{rw} is the number of times when r reads and w writes are performed in parallel in simulation, and E_{static} is the energy per t_d due to leakage current of the RF. Again, the parameters E_{rw}, E_{static}, and t_d are obtained from the cost database and U_{rw} and n_c are given by the instruction set simulator.

The energy consumed by the interconnection is obtained with the same principle as with function units in (1). For estimating the energy of a control unit, E^{CNTRL}, we have used an extremely simple model:

$$E^{CNTRL} = n_r \left(E_0 + dE_s \right) \tag{3}$$

where n_r is the total number of bits in all the registers in the control unit and E_0 the dynamic energy per clock cycle of a 1-bit register. The parameter d denotes the density of interconnection network, which is describes how many of all the possible socket connections are used, i.e., $d = 1$, if all the socket connections are connected (fully connected interconnection). The parameter E_s describes the additional energy consumed per clock cycle due to instruction decoding and driving of the control signals to a socket connection.

4 Experimental Results

In order to compare the accuracy of the proposed estimation scheme, we characterized a set of function units described in VHDL on a commercial 0.11 μ ASIC technology and created a cost database. We used a number of applications including 8×8 discrete cosine transform and Viterbi decoding and selected several target architectures for each application, with varying level of parallelism. The applications were compiled with Move tools onto the target architectures and the Move simulator was used to obtain the run-time statistics. A VHDL description of each target architecture was generated.

For each target architecture, the reference area was obtained by synthesizing the design using the Synopsys Design Compiler. The timing constraints used in the synthesis were used as values of requested clock period in the estimations. The energy references were obtained by performing first gate-level simulation for capturing the switching activity. The activity information was used in the power analysis of the Design Compiler. The simulation was performed by using the binary code generated by the Move tools.

An example of experimental results is given in Fig. 5, which shows area and energy of function units (FU), register files (RF), interconnect (IC) and control logic (CNTRL) for one application executed on three different architectures. According to preliminary experiments, the average error in area estimation is 5% and the maximum error is 18.3%. The largest error is in the interconnect and control logic. The error in energy estimation is larger: 10.5% on average and maximum of 34.8%. Again, the largest error is typically in the interconnect and control unit due to the fact that the used models are simple.

The speed of the estimation process is mainly dependent on the number of function units and register files. The size of the cost database has also an effect on the estimation speed. In order to have an idea of the estimation speed and its portion of the overall design space exploration process, we measured the time an estimation takes on a Linux work station with 600 MHz Pentium 3 processor. Target architectures ranging from 20–70 kgates required 1–5 sec while simulation of these (500–2M instruction cycles) took 2–60 seconds. The actual compilation took 2-120 seconds. Therefore, one iteration in the design space exploration, estimation of one processor took 5–180 sec. When those processors are synthesized on a work station with a 3 GHz Pentium 4 processor, the logic synthesis alone takes from 2 hours to 2 days. The power analysis of the synthesized structure with the switching activity capture in simulator requires another 2 hours to 2 days depending on the number of instruction cycles.

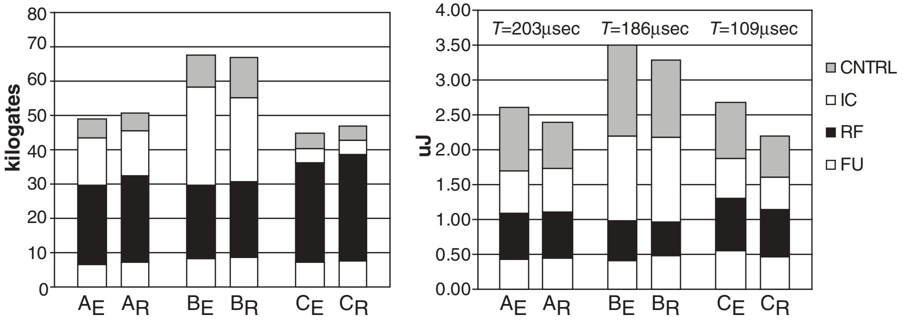


Fig. 5. Experimental area and energy comparisons of three processor architectures executing the same application. Processors *A* and *B* run at clock frequency of 100 MHz and processor *C* at 200 MHz. A_E,B_E,C_E: Estimates. A_R,B_R,C_R: References. *T*: Execution time

5 Conclusions

In this paper, we have proposed a cost estimation methodology suited for design space exploration for TTA processors. In addition, we proposed an improved exploration process with tool-assisted selection of pipelining degree for function units of a given target architecture. Our preliminary comparisons show that the proposed cost estimation process is really quick. Therefore, it is well suited for exploration where hundreds or even thousands of architectures are evaluated. In addition, the experiments show that the area accuracy of the estimation is clearly sufficient for design space exploration. The energy estimation does not perform as well, especially the estimation of interconnect could be improved. However, the energy estimation accuracy is still sufficient for exploration; it allows comparison of different processor architectures.

Acknowledgement

This work has been supported in part by the Academy of Finland under project 205743 and the National Technology Agency of Finland under project “Flexible Design Methods for DSP Systems”.

References

1. Corporaal, H., Arnold, M.: Using transport triggered architectures for embedded processor design. Integrated Computer-Aided Eng. **5** (1998) 19–38
2. Corporaal, H.: Microprocessor Architectures: From VLIW to TTA. John Wiley & Sons, Chichester, UK (1997)
3. Jin, H.S., Jang, M.S., Song, J.S., Lee, J.Y., Ki, T.S., Kong, J.T.: Dynamic power estimation using the probabilistic contribution measure (PCM). In: Proc. Int. Symp. Low Power Electronics and Design, San Diego, CA (1999) 279–281
4. Wu, Q., Qiu, Q., Pedram, M., Ding, C.S.: Cycle-accurate macro-models for RT-level power analysis. IEEE T. VLSI **6** (1998) 520–528
5. Ahonen, T., Nurmi, T., Nurmi, J., Isoaho, J.: Block-wise extraction of Rent’s exponents for an extensible processor. In: Proc. IEEE Comput. Soc. Ann. Symp. VLSI, Tampa, FL (2003) 193–199
6. Chang, N., Kim, K., Lee, H.G.: Cycle-accurate measurement and characterization with a case sudy of the ARM7TDMI. IEEE T. VLSI **10** (2002) 146–154
7. Contreras, G., Martonosi, M., Peng, J., Ju, R., Lueh, G.Y.: XTREM: A power simulator for the Intel XScale core. In: Proc. ACM Conf. Language Compilers Tools Embedded Syst., Washington, DC (2004) 115–125
8. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: A framework for architectural-level power analysis and optimizations. In: Proc. Int. Symp. Comput. Arch., Vancouver, BC, Canada (2000) 83–94
9. Vijaykrishnan, N., Kandemir, M., Irwin, M.J., Kim, H.S., Ye, W., Duarte, D.: Evaluating integrated hardware-software optimizations using a unified energy estimation framework. IEEE T. Comput. **52** (2003) 59–76
10. Gerlach, J., Rosenstiel, W.: A scalable methodology for cost estimation in a transformational high-level design space exploration environment. In: Proc. Design, Automation and Test in Europe, Paris, France (1998) 226–231
11. Talarico, C., Rozenblit, J.W., Malhotra, V., Stritter, A.: A new framework for power estimation of embedded systems. IEEE Comput. **38** (2005) 71–78

Ultra Fast Cycle-Accurate Compiled Emulation of Inorder Pipelined Architectures*

Stefan Farfeleder[1], Andreas Krall[1], and Nigel Horspool[2]

[1] Institut für Computersprachen, TU Wien, Austria
{stefanf, andi}@complang.tuwien.ac.at
[2] Department of Computer Science, University of Victoria, Canada
nigelh@uvic.ca

Abstract. Emulation of one architecture on another is useful when the architecture is under design, when software must be ported to a new platform or is being developed for systems which are still under development, or for embedded systems that have insufficient resources to support the software development process. Emulation using an interpreter is typically slower than normal execution by up to 3 orders of magnitude. Our approach instead translates the program from the original architecture to another architecture while faithfully preserving its semantics at the lowest level. The emulation speeds are comparable to, and often faster than, programs running on the original architecture. Partial evaluation of architectural features is used to achieve such impressive performance, while permitting accurate statistics collection. Accuracy is at the level of the number of clock cycles spent executing each instruction (hence the description *cycle-accurate*).

1 Introduction

Emulation of instruction sets of different architectures is common. Originally, all emulators were interpreter-based. An interpreter mimics the execution of a standard computer by repeatedly fetching an instruction, decoding that instruction, and then executing it. The implementation is straightforward and allows insertion of monitoring code into the interpreter to gather any desired statistics. SimpleScalar and some other modern simulators still use interpretation because it allows cycle-accurate emulation of all features of today's complex architectures with out-of-order instruction execution [1].

The biggest disadvantage with interpreters is their extremely slow execution speed, which can be three to five orders of magnitude slower. Improving emulation speed is clearly desirable. In this paper, we describe techniques which achieve a speed-up by about three orders of magnitude — making the emulated program on a PC faster than on the original architecture.

2 Related Work

One technique for improving emulation speeds is memoization. Micro architecture states and the resulting simulator actions are cached. Then the emulation can be "fast

* This research was supported in part by Infineon and the Christian Doppler Forschungsgesellschaft.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 222–231, 2005.

forwarded" whenever a cached state is reached. Schnarr and Larus [2] improved the speed of FastSim by 5 to 12 when emulating an architecture similar to a MIPS R10000. The speed can be further improved by using subroutine threaded interpreters which cache changed program parts [3].

Translating emulators are orders of magnitude faster than interpreters. Binary translation was first used for functional simulation of other architectures. A static binary translator takes a complete program, determines the program structure and translates the program into an equivalent one on the host architecture. Problems arise when indirect branches cannot be resolved at compile time or self-modifying code is used. A solution is to combine the translated program with an interpreter which is used in such a case. Binary translators have been successfully used for the simulation of the IBM 370 architecture [4] and for the migration of programs from the MIPS architecture to the Alpha architecture [5]. In contrast, dynamic binary translators convert short sequences of linear code into native code of the host architecture at runtime. This is the approach embodied in the Transmeta Crusoe architecture [6].

Shade [7] performs functional emulation and instrumentation, where collecting traces and similar information incurs a 2.8 - 6.1 slowdown. Embra [8] is a functional CPU model in SimOS and runs about 10 to 30 times slower by translating target instructions into the native instructions of the host. Bintrans [9] is a retargetable binary translator. From a description of the source and target architectures, a dynamic binary translator is automatically generated which executes programs between 1.8 and 2.5 times slower than the original.

Binary translation is tied to a fixed host architecture. Compiled emulation is more flexible because it generates C (or other high-level) source code for the emulated program. The compiler can optimize away most of the intermediate computations and thus improve performance. Mills et al. [10] generate one function for the complete program implementing branches by a switch statement. Amicel and Bodin [11] used assembly language source as the input language and generated C/C++ machine code. Retargetable compiled emulation has been successfully applied by Pees et al. [12].

3 The xDSPcore Processor Architecture

The simulated processor, xDSPcore [13], is a five-way variable-length very long instruction word (VLIW) load/store digital signal processor (DSP) with pipelined inorder execution. Up to five instructions are executed in each cycle. It supports some common extensions for the DSP domain, such as SIMD (single instruction multiple data) instructions, multiply-accumulate instructions, various addressing modes for loads and stores, fixed point arithmetic, predicated execution, etc. The processor's register file consists of two banks, one for data registers, the other for address registers. Each data register is 40 bits wide, but can also be used as a 32 bit register, or as two registers of 16 bit width ("shared registers", "overlapping registers", "register pairs").

The xDSPcore is a pipelined architecture. Some instructions need more than one execution stage. Register operands are read at the beginning and written at the end of the pipeline stage where they are needed. Branches have delay slots which can be filled with any instruction bundle. The xDSPcore's hardware loop instructions allow a fixed number of repetitions of a piece of code without having to manage the loop counter in the code itself.

The simulated processor can make two memory accesses per cycle if they are to different banks, otherwise an additional memory access cycle is needed. There is no data cache, but there is an instruction buffer. The instruction buffer minimizes memory accesses and thus reduces power consumption on the xDSPcore. It has eight slots. Each slot holds one fetch bundle, which consists of four instruction words, plus an *executed bit*. The executed bit is set after all four of the instruction words are executed. The slot can be recycled and its contents overwritten by another instruction bundle only after the executed bit has been set. The xDSPcore's fetch unit reads one fetch bundle per cycle and writes it in a round-robin manner to the next slot in the buffer, omitting the write if that bundle is already cached or if the buffer slot does not have its executed bit set. A second unit, the aligner unit, reads four fetch bundles from the buffer and issues a stall if an instruction word needed for the next instruction bundle is missing.

4 Simulator Details

The requirements of our simulator were:

- fastest possible execution,
- cycle and state accurate,
- debugger support (single stepping, breakpoints),
- convenient architecture specification,
- portability (should run on common 32 and 64 bit computers).

The performance and portability requirements require compiled emulation. The assembly language source of the program to be emulated is translated into an equivalent C program which emulates the whole functionality of the simulated architecture. Despite difficulties caused when emulating a pipelined parallel architecture, basic blocks and loops are used as translation units. To handle unpredictable computed jumps and to support debugging, a full interpreter is integrated with the compiled emulator. Control is passed back and forth between the two components as required. The interpreter has a GUI which displays assembler source, and supports single-stepping and breakpoints.

For extending the architecture and for easy retargeting to other architectures, the syntax and semantics of the instruction set are specified in a XML configuration file. In the following sections, we describe how various problems in the emulator are solved.

4.1 XML Configuration File

Both the interpreter and the compiled emulator read their configurations from an XML file. It describes the complete instruction set and the hardware configuration for the register file, the pipeline, the instruction buffer, etc. The description of an instruction includes the execution semantics and additional text used for automatic documentation generation and to describe calling conventions. Figure 1 shows a slightly simplified and edited version of the XML description of the `ld` (Load) instruction. The instruction reads the value of an address register at the beginning of stage `EX1`, adds 2 to the register

```
<instruction>
  <mnemonic>ld</mnemonic>
  <operands>
    <operand>ADDR_REG</operand>
    <operand>LX_DX_RX_REG</operand>
  </operands>
  <syntax>(op1)+, op2</syntax>
  <semantics>
    <execute>READ_OP1</execute>
    <execute>MOD_OP1</execute>
    <execute>MEM_READ</execute>
    <execute>WRITE_OP2</execute>
  </semantics>
</instruction>
```

```
<map key="READ_OP1">
  <timing>EX1,begin</timing>
  <code>tmp1 = %op1</code>
  <code>tmp2 = %op1 + 2</code>
</map>
<map key="MOD_OP1">
  <timing>EX1,end</timing>
  <code>%op1 = tmp2</code>
</map>
<map key="MEM_READ">
  <timing>EX2,begin</timing>
  <code>tmp3 = mem[tmp1]</code>
</map>
<map key="WRITE_OP2">
  <timing>EX2,end</timing>
  <code>%op2 = tmp3</code>
</map>
```

Fig. 1. `ld` instruction with timings in the XML file

at the end of `EX1`, uses the old value as the address for a memory read at the beginning of stage `EX2` and stores the read value into another register at the end of the stage.

The identifiers within the `<execute>` elements reference other places in the XML file (shown in Figure 1), where the timings and the code that has to be generated for such an instruction part are stored. This separation of concerns facilitates maintenance – since many instructions share common parts, changes can be made at a single place.

The `<operands>` and `<syntax>` elements shown in Figure 1 are used for the assembler front-end. After an assembler line is split into simple tokens, checks are made as to whether the syntax and the types of the operands match the information found here.

4.2 Dividing the Instruction Bundles into Basic Blocks

The instruction bundles are traversed to find all basic block leaders. A leader is an instruction bundle that meets one or more of the following requirements:

1. it is a target of a branch instruction,
2. it starts the body of a hardware loop, or
3. it follows a branch instruction or the end of a hardware loop body.

For those branch instructions that have a branch delay, the instructions in the branch delay slots are appended to the branch instruction's basic block. If an additional branch is executed in a branch delay slot, only the first instruction of the target basic block is executed. In this case, a duplicate basic block which contains only the first instruction is generated. Each of these basic blocks is translated into a single C function in the generated output. This keeps the functions small, resulting in short compilation times and good optimization by the C compiler.

EX1	begin	`tmp1 = r0`
		`tmp2 = r0 + 2`
	end	`r0 = tmp2`
EX2	begin	`tmp3 = mem[tmp1]`
	end	`l0 = tmp3`

Fig. 2. Code for `ld (r0)+, l0`

4.3 Generating Code for Instructions

Consider an actual instruction with real operands, like `ld (r0)+, l0`. The placeholders for the operands that were shown in Figure 1 are simply filled with the actual operands. Figure 2 depicts the code generated for this instruction. The identifiers starting with `tmp` in the table are temporary variables used to cache register values or computed values. The C compiler should optimize unnecessary copies away. These temporaries also solve interdependencies between different pipeline stages of overlapping instructions in an elegant way.

Many arithmetic instructions can be implemented by a single C operator. Other instructions like multiply-accumulate, bit insertion or saturated computations do not have direct C counterparts. They are implemented by groups of operations or small inline functions which are read from the XML file.

4.4 Control Flow

Each generated C function returns the number of the next basic block to be be executed. This number is used as an index into an array of function pointers to locate the next basic block's function. The compiled simulator's main loop has the following simple structure:

```
int bbnr = <number of starting block>;
while ((bbnr = bbptr[bbnr]()) >= 0) ;
```

A software stack simulates the hardware stack for subroutine calls. At a call, the number of the basic block following the call instruction is pushed onto the stack, the called function number is returned and is thus executed next. A return instruction pops a function number from the stack and returns it.

4.5 Instructions Crossing Basic Block Boundaries

Consider the assembler code show in Figure 3. Because the `EX2` stage of the `ld` instruction is executed at the same time as `movr`'s `EX1` stage and because register `l0` is written at the end of a cycle, register `l1` receives `l0`'s old value. Therefore executing the whole `ld` instruction at the end of the basic block which contains the `br` instruction would give wrong results. To resolve these conflicts, the code fragments of `ld`'s `EX2` stage are moved into the basic block that begins with the label `foo:` and will be executed there in the correct order. The decision whether those moved code parts need to be executed is determined by a global variable that remembers the last executed basic block.

Basic blocks can be duplicated to improve performance. For every predecessor P_i of basic block B which has leftover pipeline stages, a specialized version B_i of basic block

```
 br foo
 nop
 ld (r0)+, l0
 ...
foo:
 movr l0, l1
```

Fig. 3. Overlapping between `ld` and `movr`

B is generated. It includes the code for the leftover pipeline stages. A global simulator switch determines the code generation scheme. In the previous example, the basic block is duplicated. Only one of them executes the second part of `ld`.

4.6 Simulating the Instruction Buffer

The addresses of the currently cached fetch bundles are stored in an array, as are the executed bits. At the beginning of each bundle, an attempt is made to insert the next fetch bundle's address into the array. A second table is used for a reverse-lookup because simulating the fully associative lookup would require up to eight comparisons per check. This second table associates each possible fetch bundle address with an index into the address array.

All instruction words between the program counter and the fetch counter are always held in the instruction buffer. Thus if one knows that the fetch counter is ahead of the instruction pointer by a sufficient amount, the check whether the instruction words needed for the execution of the next bundle are available can be omitted. To simulate this statically, the following strategy is applied. The program counter is initially set to the address of the first instruction bundle and the fetch counter is set to the address of the first fetch bundle. Program flow is simulated by adding four to the fetch counter and the amount of memory used by the instruction bundle to the program counter at every step. If the fetch counter does not exceed the program counter, there is no guarantee that the bundle is in the buffer. In this case, extra code is generated which performs a look-up for the needed address and to simulate a stall if it could not be found.

As already stated, executing a branch instruction sets the fetch counter and all executed bits. Code to simulate these actions is executed at the start of the destination basic block. When that destination block can be reached by both branching and by sequential execution, two versions of the block are compiled — one with and one without the extra code to set the fetch counter and the executed bits. Finally code to set the executed bit in the instruction buffer is inserted after all instruction words of a fetch bundle are executed.

Simulating the instruction buffer is expensive. Techniques to decrease the costs by computing extensive lookup tables at compile time are being explored.

4.7 Hardware Loops

The loop instruction is simulated by pushing a function pointer to the loop body's first basic block and the iteration count onto a stack. At the end of the loop, the counter is decremented; if it reaches zero, the following basic block gets executed, otherwise execution continues with the beginning of the loop body as found on the stack.

If a hardware loop consists of a single basic block, the simulator optimizes the loop into a C `for(;;)` statement, thus eliminating the overhead caused by a function call for each iteration and enabling the C compiler to apply further optimizations. If a hardware loop is sufficiently small to fit into the instruction buffer, a different optimization can be performed. The loop body is unrolled three times; the first copy simulates the buffer as described in the previous section for the first iteration, the second one repeats the body $n-2$ times. Since the instruction words are already buffered, the fetch simulation can be completely omitted. Finally the third copy of the body simulates the last iteration of the loop.

4.8 Memory Stalls

The xDSPcore has two memory ports, the X port covering the lower half of the data memory and the Y port covering the upper half. Two memory accesses are possible in a single cycle only if they do not use the same port, otherwise a pipeline stall occurs and the second access is deferred to the next cycle.

If two memory accesses are detected in a bundle, code to test whether the two memory addresses use the same port has to be inserted. If `tmp1` and `tmp2` are temporary variables holding the values of two address registers that are used to access memory, then the code to check if a stall occurs is similar to this:

```
if (!((tmp1 ^ tmp2) >> 15)) {
    ... /* issue a stall */
}
```

4.9 Collected Statistics

Each basic block has an associated counter which has to be incremented at runtime when entered. Using these counters, the dynamic number of executed instructions, bundles, the average number of instructions in a bundle, the frequency of each instruction, etc., can easily be computed. The number of memory stalls and aligner stalls are also counted. In addition, the emulator maintains extra counters for `.PROFILE` pseudo-instructions that are generated by the C compiler. They are used for feedback-driven optimization.

5 Experimental Results

Six sample programs, which represent typical applications for the xDSPcore processor, were used in our experiments: blowfish (symmetric block ciphering), dct8x8/dct32 (discrete cosinus transformations), g721 (voice compression), serpent (cryptographic algorithm) and viterbi (Viterbi decoder). The sizes of these programs and other characteristics are listed in Table 1. The dynamic parallelism column shows the average number of instructions executed in each cycle. The parallelism and the dynamic average basic block length have a significant effect on how efficiently the program can be emulated.

The left part of table 2 shows the speed of the six programs on a simple interpreter. Because statistics gathering has such a large effect on emulation speed, the speed is

Table 1. Characteristics of Test Programs

	Source size	Object size	Dynamic parallelism	Average basic block length
blowfish	25.8 kB	32 kB	1.91	14.38
dct8x8	43.9 kB	7 kB	1.85	7.48
dct32	35.8 kB	34 kB	2.14	8.73
g721	28.5 kB	5 kB	1.29	6.57
serpent	144.1 kB	46 kB	1.68	8.31
viterbi	36.6 kB	23 kB	1.21	216.85

Table 2. Emulation Speeds with an Interpreter and Compiler

	interpreted		compiled	
	with statistics	without statistics	with instr. buffer	without instr. buffer
blowfish	.083 MHz	.207 MHz	165 MHz	302 MHz
dct8x8	.082 MHz	.205 MHz	95 MHz	190 MHz
dct32	.071 MHz	.187 MHz	105 MHz	204 MHz
g721	.078 MHz	.198 MHz	78 MHz	259 MHz
serpent	.040 MHz	.208 MHz	120 MHz	258 MHz
viterbi	.094 MHz	.214 MHz	181 MHz	566 MHz

Table 3. Resources Needed to Create the Compiled Simulation

	Generation time (s)	Compile time (s)	C code size (kB)	Binary size (kB)
blowfish	3.22	3.06	316	257
dct8x8	3.32	4.51	421	396
dct32	3.27	5.13	780	542
g721	4.97	7.47	454	404
serpent	9.41	24.36	2081	1518
viterbi	3.50	60.85	475	411

shown with statistics gathering enabled and disabled. The right part of table 2 shows the execution speed of each of the programs when emulated with *Compiled Emulation*. The two columns show the cost of emulating the instruction buffer of the xDSPcore architecture. However it is necessary for guaranteeing cycle-accurate performance statistics. Statistics gathering has negligible effect on timings for the compiled emulation. Therefore, separate timing data is not shown for this case in the table.

The effective speed-up through using the compiled technique versus interpretation can be estimated by comparing the numbers in the “with statistics” column of Table 2 with the numbers in the “with instruction buffer” column of Table 2. The speed-ups range from 1000 to 3000. It can be seen that the largest speed-ups occur for the programs which have the longest basic blocks.

Finally, Table 3 shows the resources needed to generate and compile the emulated programs. Although the compiled programs are much larger than the original programs on the xDSPcore platform, it should be remembered that they are executed on a much more powerful computer where memory is not a limitation. All measurements were made on an AMD Opteron 2Ghz CPU. The C code was translated by the Intel compiler with the `-O3` optimization level.

6 Conclusion

We have presented a novel approach for retargetable emulation of an architecture with some challenging features which include pipelining, a VLIW design, banked memory and an instruction cache. By generating C code which represents a translation of the original program at the basic block level, and which embodies the particular features of the emulated architecture, we have achieved impressive performance results. To our knowledge, we are the first to exploit partial evaluation of emulated features and extensive code duplication of the emulated program. The emulation speed is up to 3000 times faster than an interpreter while still maintaining a faithful simulation of the original architecture down to the number of clock cycles consumed.

References

1. Austin, T., Larson, E., Ernst, D.: SimpleScalar: An infrastructure for computer system modeling. Computer **35** (2002) 59–67
2. Schnarr, E., Larus, J.: Fast out-of-order processor simulation using memoization. In: Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII), ACM SIGPLAN, ACM (1998) 283–294
3. Nohl, A., Braun, G., Schliebusch, O., Leupers, R., Meyr, H., Hoffmann, A.: A universal technique for fast and flexible instruction-set architecture simulation. In: Proceedings of the 39th conference on Design automation, ACM Press (2002) 22–27
4. May, C.: Mimic: a fast system/370 simulator. In: Papers of the Symposium on Interpreters and interpretive techniques, ACM Press (1987) 1–13
5. Sites, R.L., Chernoff, A., Kirk, M.B., Marks, M.P., Robinson, S.G.: Binary translation. Communications of the ACM **36** (1993) 69–81
6. Dehnert, J.C., Grant, B.K., Banning, J.P., Johnson, R., Kistler, T., Klaiber, A., Mattson, J.: The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO '03). (2003)
7. Cmelik, B., Keppel, D.: Shade: A fast instruction-set simulator for execution profiling. ACM SIGMETRICS Performance Evaluation Review **22** (1994) 128–137 Special Issue on Proceedings of the 1994 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '94; 16–20 May 1994; Vanderbilt University, Nashville, TN, USA).
8. Witchel, E., Rosenblum, M.: Embra: Fast and flexible machine simulation. In: Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. Volume 24,1 of ACM SIGMETRICS Performance Evaluation Review., New York, ACM Press (1996) 68–79
9. Probst, M.: Dynamic binary translation. In: UKUUG Linux Developer's Conference 2002. (2002)
10. Mills, C., Ahalt, S.C., Fowler, J.: Compiled instruction set simulation. Software – Practice and Experience **21** (1991) 877–889

11. Amicel, R., Bodin, F.: A new system for high-performance cycle-accurate compiled simulation. In: 5th International Workshop on Software and Compilers for Embedded Systems. (2001)
12. Pees, S., Hoffmann, A., Meyr, H.: Retargetable compiled simulation of embedded processors using a machine description language. ACM Transactions on Design Automation of Electronic Systems. **5** (2000) 815–834
13. Krall, A., Hirnschrott, U., Panis, C., Pryanishnikov, I.: xDSPcore: A Compiler-Based Configurable Digital Signal Processor. IEEE Micro **24** (2004) 67–78
14. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer **35** (2002) 50–58

Generating Stream Based Code from Plain C

Marcel Beemster, Hans van Someren, Liam Fitzpatrick, and Ruben van Royen

ACE Associated Compiler Experts bv,
De Ruyterkade 113, 1011 AB Amsterdam, The Netherlands
marcel@ace.nl
http://www.ace.nl

Abstract. The Stream model is a high level Intermediate Representation that can be mapped to a range of parallel architectures. The Stream model has a limited scope because it is aimed at architectures that reduce the control overhead of programmable hardware to improve the overall computing efficiency. While it has its limitations, the performance critical parts of embedded and media applications can often be compiled to this model. The automatic compilation to Stream programs from C code is demonstrated.

1 Motivation

Now that humanity has learned to make very fast processors, it turns out that they are not always efficient enough to fit our desire for portability. Fortunately there is a lot of room for improvement. Programmable microprocessors - including RISC, CISC, VLIW and DSP flavors - are very inefficient. For every data operation an instruction has to be decoded, control signals sent, operands retrieved from a multi-ported (expensive) register file, data moved to the functional unit, computed and finally moved back to the register file. This does not even take speculative actions, re-order buffers, bypasses, prediction mechanisms and many other techniques into account.

Nevertheless, programmable microprocessors remain extremely popular in many computing domains, including that of embedded computing where—by nature—the application is more static than applications that run on, for example, a PC. The primary reasons are flexibility and sequential programmability.

Flexibility—due to software programmability—is important when time to market, emerging standards, and product diversity are important. When a short time to market is a goal, product specifications are often not available until the project is well underway. Software provides the flexibility to adapt the product. Emerging standards pose similar constraints. In-field upgrades are only possible with programmable devices.

Putting relatively small extensions aside, almost all programmable microprocessors present a sequential programming model that is suitable for programming in assembly or a sequential high level language such as C. Sequential programmability is important because parallel programming is difficult and non-portable.

Parallelism holds the key to solving many of the inefficiencies of the sequential processor. It has done so for many years already—right from the beginning of programmable computing. Parallel hardware is very easy to build. If parallel programming were not so hard, parallel computing would be much more common than it is today.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 232–241, 2005.

The kind of parallel hardware that is used in embedded systems often has the goal of reducing power consumption in addition to speeding up the application. A simple form is the extension of processors with SIMD instructions operating on wide data paths. Such an instruction operates, for example, on four data values at a time and saves the control overhead of three individual values.

This paper discusses a stream based, abstract, parallel model of computation and how to extract it from a subset of standard C. The aim of the model is to serve as an intermediate representation in the compilation to programmable parallel architectures. The model is an abstract model—it requires further mapping to the architecture to deal with timing and resource constraints. For this reason, the model is rather restricted and very static. This helps in the mapping to different kinds of parallel architectures, each with its own special features and limitations. The static nature of the model implies that much of the control overhead can be removed and that parallelism at the hardware level is readily extracted.

The static nature of the model implies that not all of standard C can be expressed in it. To compile arbitrary C programs requires a hybrid architecture that has an efficient parallel data path as well as a fully flexible sequential data path. A compiler is expected to keep track of which parts of the program are mapped to which data path.

The Stream model described here can be contrasted to the Streams-C high level language [1]. Streams-C is a programming language, not an IR, and has a much richer computational model including independently running processes. It is developed specifically for programming FPGAs. Although both languages are based on the data flow principle, the differences in design goals result in two very different languages.

The goals of StreaMIT [2] are closer to that of the Stream model. An important difference is that the filters in StreaMIT can have arbitrary complexity, while the Stream model's computational elements are standard primitives. Timing and communication are much less restricted in StreaMIT, even allowing for asynchronous messages. Still, many of the optimization and mapping algorithms described for StreaMIT also apply to this work.

2 The Stream Model

The Stream processing model that is introduced here is designed in the context of the commercial CoSy compiler development system [3]. CoSy is used by many companies and research institutes to create compilers for processors that are used in embedded applications. CoSy is a compiler development system—it is highly flexible and can be adapted to build compilers for a wide variety of processors. Compilers built with CoSy are most often C compilers.

The Stream model is described here together with its derivation from standard C. It serves as an intermediate representation in the compiler and requires additional mapping to actual hardware.

Figure 1 shows a graphical representation of an example Stream program. The Stream model is always a directed data flow graph, no cycles are possible. A given instance of the model consists of a number of connected elements, each implementing a specific function that may be different from other elements. At the input side of the

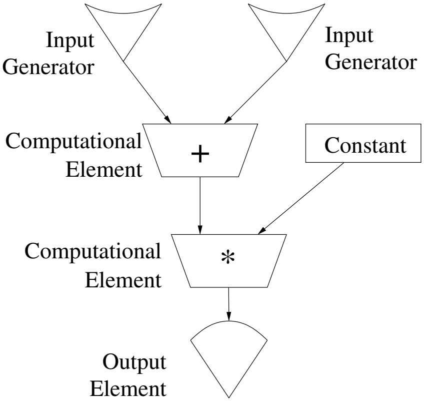


Fig. 1. A graphical representation of an example Stream program

graph are input and constant elements. At the output side are one or more output elements. Between input and output elements there is a directed network of computational elements.

Input elements read from memory, output elements write to memory. The abstract Stream model executes one *iteration* at a time. The term iteration is used since it corresponds directly with one iteration of the inner loop of the original C source code. At every iteration input values are produced, then pass through the computational elements network and the resulting values written to memory by the output elements.

The input for translation to the Stream model are C loop nests. These nested loops are the source of the parallelism that can be expressed in the Stream model. The rest of the program must be mapped to the sequential data path using traditional compiler techniques.

2.1 Input Generators

Input generators arise from array access expressions. The following code is an excerpt from a matrix multiply operation:

```
for (i=0;i<N;i++){
   for (j=0;j<N;j++){
      for (k=0;k<N;k++){
            ... a[i][k] ...
}  }  }
```

The expression `a[i][k]` corresponds to a Stream input generator. When executing the loop in its sequential form it generates a stream of values from the array `a`. This stream is described in the Stream model as follows:

```
s = StreamInput3( &a[0][0], N, N,     /* Outermost */
                            N, 0,     /* Middle    */
                            N, 1 )    /* Innermost */
```

This is a 3-dimensional stream generator with the base address a (&a[0][0]). It is three dimensional because the source expression is inside three nested loops. Each dimension is described by a tuple with the number of iterations and the stride. The StreamInput3 primitive generates a stream of values obtained from the array by nested address generation of the three dimensions, where the innermost dimension runs with the highest frequency.

The three tuples are derived by analyzing the use of the iteration counters i, j and k in the expression a[i][k]. Each iteration loops N times. This corresponds to the initial value N in every tuple. The second value is the stride with which each iteration walks through the array. The i iteration counter is used in the "big" dimension of the two dimensional array. Every step accesses the following row and each row has a size of N. So the stride for the outermost dimension is N. The iteration counter j is not used in the array expression and so does not contribute to the address generation. Its stride is therefore 0. Finally the k iterator is used in the "small' dimension of the expression. Hence the innermost stride is 1.

The result is an input element that generates a stream of N*N*N data values. At every iteration of the model, one data value is taken off every input stream.

Output elements are very similar to input elements and correspond to array-element assignment in the source program. (See example below.)

A number of different input primitives correspond to different levels of loop nesting. Input address generation is limited to affine expressions of iteration variables. It is not possible to use data dependent values in the array indexing expressions. With this limitation the address generation in the input and output generation is completely decoupled from computations in the data flow graph. This decoupling is important because it makes the control flow of the model completely independent of the data values that are computed.

2.2 Computations on Streams

Here is a more complete example:

```
for( i = 0 ; i < 64 ; i++ ) {
    t    = a[i] + b[i] ;
    z[i] = t    + c[i*2] ;
}
```

In the compilation from a C loop nest to a Stream program, every expression inside the loop represents a stream with one data value for every iteration. Covering the code with streams is a bottom up process, starting at the leafs. In the expression a[i] + b[i], each of a[i] and b[i] represent an input stream. The two streams are combined with the + operator, resulting in another stream. This stream is bound to the variable t by the assignment. The t stream is used again in the next statement.

The complete Stream program for the example is:

```
Stream as = StreamInput1( &a[0], 64, 1 )
Stream bs = StreamInput1( &b[0], 64, 1 )
Stream ts = StreamAddition( as, bs )
Stream cs = StreamInput1( &c[0], 64, 2 )
Stream zs = StreamAddition( ts, cs )
StreamOutput1( zs, &z[0], 64, 1 )
```

3 Extending the Basic Model

The basic Stream model described so far is very simple. The data flow graph is directed and static, every operation completes within the iteration, there is no conditional execution and there is no state inside the graph.

The graph must remain directed and static. This enables the mapping onto a wide range of parallel processor architectures. But many other limitations can be lifted in order to facilitate more complex programs to be expressed.

3.1 Conditional Execution

Adding conditional execution to a data flow model is relatively easy. Consider:

```
if( c > 0 ) { a = b + 2 ; }
else        { a = b ; }
```

To implement this, the `StreamCondition` element is introduced. It connects to three input streams.

```
Stream StreamCondition( Stream cs, Stream ts, Stream fs )
```

The first input stream is the one that controls the condition. It is a stream of boolean values with the results of `c>0` for every iteration. The other two input streams are for `b+2` (true values) and `b` (false values). This `StreamCondition` takes one value of every Stream and selects either the value from `ts` or `fs` depending on the value from `cs`.

Note that this element makes the selection after both the `then` and the `else` branch of the original code have been computed. In a sequential processor the choice would have been made beforehand, thus saving work.

In the Stream model late selection is unavoidable. The formalism does not allow non-determinism: a stream that may or may not produce a value at a given cycle, depending on a computed value cannot be dealt with. In the mapping to hardware, this restriction may be lifted.

3.2 Reduction Operators

Reduction operators such as accumulation are common in signal processing applications:

```
for( i = 0 ; i < 64 ; i++ ) {
    accu = 0 ;
    for( j = 0 ; j < 32 ; j++ ) {
            accu = accu + x[i][j] * coeff[j] ;
    }
    result[i] = accu ;
}
```

In this example the multiply-accumulate is done once for every inner loop iteration. But the accumulated value is written (only) once every outer loop iteration. This is the nature of a reduction operation: the rate at which values are produced becomes lower.

In the Stream model the accumulator is included as a primitive operation. It has two operands, an input stream and a parameter *rate*. The rate determines how many values are accumulated before a result value is produced. In the example above the rate is 32:

```
multRes = StreamMultiply( xIn, coeffIn )
accus = StreamAccumulate( multRes, 32 )
```

In this example the result Stream `accus` has 32 times fewer elements than the input stream `multRes`. In the data flow graph this means that on the `multRes` edge of the graph there is an element in every iteration, but on the `accus` stream there is only an element at every 32nd iteration. The frequency of elements on certain edges is reduced, but in the model it is still known exactly at what iterations the elements occur.

Note that the `StreamAccumulate` primitive introduces state in the data flow graph. The accumulator node carries two values—the accumulated value, by definition initialized at 0, and a counter—from one iteration to the next.

3.3 Reusing Values

Consider the following loop:

```
for( i = 1 ; i < 64 ; i++ ) {
        b[i] = a[i] + a[i-1] ;
}
```

In a straightforward translation to the Stream model this would give two input streams reading from memory and one output stream. But that would be quite wasteful: the array `a` would be read twice and memory access is so expensive that that should be avoided. The observation with this loop is that every value `a[i]` is used again one iteration later as `a[i-1]`.

In the Stream model this requires a node that delays a value for use in the next iteration. The two input streams of the example code can then be created with:

```
aStream = StreamInput1( &a[1], 64, 1 )
aSub1Stream = StreamDelay1( aStream, a[0] )
```

The function of the `StreamDelay1` operator is to delay every value in its input stream until the next iteration. At the very first iteration it is initialized to produce the value `a[0]`.

The same mechanism is also used when temporary variables are used to take values to the next iteration.

For the delay operator to be useful requires a fixed distance between the iteration in which the value is produced and in which it is used. In the example it is one, but larger distances are possible. In that case, more than one initial value is needed to initialize the delay pipeline.

4 Compiling a Real Application

The Discrete Cosine Transform lies at the heart of many image compression algorithms and is representative of many of the media-processing applications that find their way into embedded devices. Its implementation must be extremely efficient to be used in hand-held devices. It is also a good example of the kinds of algorithms that are used in this domain. The code embeds a significant amount of parallelism, but to exploit that from the original C code is non-trivial, let alone to map it to a specific parallel architecture.

The code below is a DCT-transform of a full image, divided into blocks. The number of blocks is 4 (NUMOFBLOCKS) and the block size is 8 (BS). The code is a 5-deep nested loop. The macros MULTIPLY and (DE)SCALE are used to implement fixed point arithmetic.

```
for (block = 0; block < NUMOFBLOCKS; block++) {
    for (y = 0; y < BS; y++) {
        for (x = 0; x < BS; x++) {
            sum = 0;
            for (v = 0; v < BS; v++) {
                for (u = 0; u < BS; u++) {
                    int tmp;
                    tmp = MULTIPLY(cos_table[x][u],
                                   cos_table[y][v]);
                    tmp = DESCALE(tmp);
                    tmp = MULTIPLY(tmp, pInput[block][v][u]);
                    sum += tmp;
            }    }
            sum = (sum >> 2) + SCALE(128);
            sum = DESCALE(sum);
            if (sum > 255)    pOutput[block][y][x] = 255;
            else if (sum < 0) pOutput[block][y][x] = 0;
            else              pOutput[block][y][x] = sum;
}   }   }
```

This code is accepted by the CoSy based Stream model generator and generates the—surprisingly concise–Stream program below. The compiler includes patterns for the recognition of accumulator operations and saturation.

```
in0 = StreamInput5(pInput,    4,256, 8,0,  8,0,
                                     8,32, 8,4)
in1 = StreamInput5(cos_table, 4,0,   8,0,  8,32,
                                     8,0,  8,4)
in2 = StreamInput4(cos_table, 4,0,   8,32, 8,0,  8,4)
calc0 = StreamMultiply(in1, in2)
calc1 = StreamAddition(calc0, 8192)
calc2 = StreamShiftright(calc1, 14)
calc3 = StreamMultiply(in0, calc2)
```

```
calc4 = StreamAccumulate(calc3, 64
calc5 = StreamShiftright(calc4, 2)
calc6 = StreamAddition(calc5, 2097152)
calc7 = StreamAddition(calc6, 8192)
calc8 = StreamShiftright(calc7, 14)
calc9 = StreamSatCeiling(calc8, 255)
calc10 = StreamSatFloor(calc9, 0)
StreamOutput3(calc10, pOutput, 4,256, 8,32, 8,4)
```

5 Mapping to Parallel Architectures

The Stream model is a high level model. Its power lies in that it can be mapped efficiently to a wide range of parallel architectures such as SIMD instruction set extensions, Vector architectures, FPGAs and Reconfigurable architectures and even SPMD machines. This is unlike standard C code, which is highly sequential by nature.

A Stream program has fully static control flow. At every point in the computation it is exactly known which value is computed, where it comes from and to where it should go. Only the values that are computed are dynamic. This static control model is exploited in the mapping to actual hardware.

5.1 FPGAs and Reconfigurable Architectures

The most natural match for the Stream model are reconfigurable architectures. Such architectures consist of a number of computational elements that can be connected through a configurable network. Generic FPGAs can be programmed like this but more particular examples are the PACT XPP [4] and the IPFlex DAP/DNA [5]. A mapping to an FPGA requires similar techniques.

Even for these architectures a mapping of the Stream model to time and resources is required.

Mapping starts with the address generators. In the DCT example five dimensional address generators are used. These are typically not available in hardware. It requires that one or more of the outer level dimensions are peeled off and moved to software control to reprogram the address generators at the right time.

Also the DCT example has two `cos_table` streams with a one-but-innermost tuple with a 0 stride. This implies that the innermost dimension values are read 8 times each. Possibly there is a way to cache these values.

Time Mapping. In the abstract model, each iteration completes before the next starts. This is not necessary, since the data flow graph is directed and fully deterministic. By keeping count of the delay of each compute element and inserting delay cycles at the right places, a fully pipelined mapping of the Stream model can be created.

Alternatively, an implementation can use synchronous transfer of values between compute elements. In that case explicit delay nodes are not needed at the expense of additional control signals between the elements.

Resource Mapping. A reconfigurable processor will have limited resources and a limited connection network. If a Stream program is too large to fit the reconfigurable processor it has to be cut. Instead of putting the complete Stream program on the architecture, it is only partially mapped. At the points where the cuts appear, output elements and input elements are inserted that store and load the data streams to temporary arrays. Since the models are fully static, computing the temporary array sizes is straightforward.

Cutting can be optimized by choosing a cutting plane with a minimal size of the temporary arrays.

5.2 SIMD Extension and Vector Architectures

SIMD Extension and Vector architectures cannot compute a whole stream program at once. The goal in mapping to these architectures is to optimize the partitioning of the streams and to keep stream communication between compute elements in (vector) registers.

Partitioning is necessary since a typical vector in these architectures is from 4 to 64 elements long. The flow of computation must be mapped to push that many elements at a time through the Stream program. At any time a static number of values will be active and these can be kept in machine registers.

Special care must be taken of the StreamDelay elements: they can be mapped onto shift or shuffle operations to shift state from one block to the next. Generating shuffle operations when starting with the Stream model is relatively easy, while they are hardly used by more traditional methods of vectorization [6].

5.3 Mapping Conditional Nodes

In the Stream model, conditional computation is implemented by selecting the right value at the end of computing all variants. It keeps the Stream model deterministic. But in an actual implementation, this may not be the most efficient because also the discarded value is computed. If an implementation permits such choices can be moved forward in the data flow network, thus saving wasted work.

For SIMD and Vector machines this is not an option. Since iterations are grouped to fit the size of the vectors, distinct variants cannot get an individual treatment, they must all follow the same flow of computation.

6 Conclusion

Above all, the Stream model is a high level intermediate representation. It is not a machine model by itself—its goal is to allow for a relatively easy mapping to parallel architectures. This is a different approach from traditional vectorizing compilers which do extensive dependence analysis and loop reorganization, but are often specific for one architecture type.

The paper demonstrates how a compiler can compute a Stream program from nested loops. The compiler does not target arbitrary C code, control flow must be static and data dependences must have a fixed distance. In embedded and media processing, a large body of performance critical code has this structure. The data flow oriented nature of these applications fits well to the Stream model.

Forced by power hungry applications, parallelism will make its way into programmable processors aimed at embedded computing, despite their awkward and non-portable programming models. The Stream model makes a modest attempt—because it has its limitations—in providing a generic intermediate representation that can be mapped to a range of parallel architectures while allowing these architectures to be programmed with a sequential programming model.

References

1. Gokhale, M.B., Stone, J.M., Arnold, J., Kalinowski, M.: Stream-oriented fpga computing in the streams-c high level language. In: FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, Washington, DC, USA, IEEE Computer Society (2000) 49–58
2. Thies, W., Karczmarek, M., Gordon, M., Maze, D., Wong, J., Hoffmann, H., Brown, M., Amarasinghe, S.: A common machine language for grid-based architectures. SIGARCH Comput. Archit. News **30** (2002) 13–14
3. Alt, M., Assmann, U., van Someren, H.: Cosy compiler phase embedding with the cosy compiler model. In: CC '94: Proceedings of the 5th International Conference on Compiler Construction, London, UK, Springer-Verlag (1994) 278–293
4. PACT XPP Technologies, Germany. `http://www.pactxpp.com` (2005)
5. IPFlex, Japan. `http://www.ipflex.com` (2005)
6. Krall, A., Lelait, S.: Compilation techniques for multimedia processors. International Journal of Parallel Programming **28** (2000) 347–361

Fast Real-Time Job Selection with Resource Constraints Under Earliest Deadline First[*]

Sangchul Han[1], Moonju Park[2], and Yookun Cho[1]

[1] School of Electrical Engineering and Computer Science,
Seoul National University, Seoul, Korea
{schan, cho}@ssrnet.snu.ac.kr
[2] Mobile Hanset R&D Center, LG Electronics, Seoul, Korea
moonjupark@lge.com

Abstract. The Stack Resource Policy (SRP) is a real-time synchronization protocol suitable for embedded systems for its simplicity. However, if SRP is applied to dynamic priority scheduling, the runtime overhead of job selection algorithms could affect the performance of the system seriously. To solve the problem, a job selection algorithm was proposed that uses a selection tree as a scheduling queue structure. The proposed algorithm selects a job in $O(\lceil log_2 n \rceil)$ time, resulting in significant reduction in the run-time overhead of scheduler. In this paper, the correctness of the job selection algorithm is presented. Also, the job selection algorithm was implemented in GSM/GPRS handset with ARM7 processor to see its effectiveness on embedded systems. The experiments performed on the system show that the proposed algorithm can further utilize the processor by reducing the scheduling overhead.

1 Introduction

In many real-time applications on embedded systems, real-time jobs communicate with each other using shared resources. In such environments, a form of synchronization protocol is necessary to control accesses to shared resources, and resources are granted to jobs and used in a mutually exclusive manner.

There are many research results on real-time synchronization protocols. Sha, Rajkumar and Lehoczky [1] proposed the Priority Inheritance Protocol and the Priority Ceiling Protocol (PCP) to solve the resource sharing problem in the fixed priority scheduling. Chen and Lin [2] extended PCP so that it can be used in Earliest Deadline First (EDF) scheduling [3]. Jeffay proposed the Dynamic Deadline Modification [4] where the deadlines of jobs are modified to avoid undesirable preemptions while accessing shared resources. The common approach of theses protocols is blocking a job at the moment it requests a resource that is held by another job. This approach makes resource sharing costly. Blocking a job causes the runtime overhead of maintaining a list of blocked jobs, additional memory consumption and context switch.

[*] This work is supported in part by Brain Korea 21 project and in part by ICT.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 242–250, 2005.

The Stack Resource Policy (SRP) [5] is another real-time synchronization protocol. It requires that a job should not be allowed to start execution if the job needs a shared resource that is held by another job. The job is not allowed to start execution until the resource is released (this is called *early blocking*). When the resource is released the job is allowed to start execution, and it is guaranteed that it will never be blocked. Therefore, under SRP, there is no actual blockage and no additional context switch. A scheduler just does not select a job that needs a resource in use currently even if the priority of the job is highest. With these features, SRP has been considered to be simpler to implement than other protocols, and to be suitable for embedded systems with limited processor power and memory capacity.

Early blocking can be realized by the CPU scheduler. Under SRP, the scheduler should select a job that satisfies the following two conditions (we call the job *most eligible*); its preemption level is higher than the system ceiling and its priority is highest among such jobs. In dynamic priority scheduling such as EDF, the conventional job selection algorithms can find the most eligible job in $O(n)$ time as we will show in Section 2. This runtime overhead becomes unendurable for embedded systems as the number of jobs increases.

In [6], a scheme is proposed that uses a selection tree as a scheduling queue (or ready queue) structure and a job selection algorithm that can find the most eligible job in $O(\lceil log_2 n \rceil)$. In this paper, we present the correctness proof of the algorithm proposed in [6], how the scheme is effective for embedded systems, and the impact of the scheme on embedded systems. We measured the runtime overhead of the scheduler on ARM7 TDMI processor which is embedded in GSM/GPRS mobile phone. The experimental results show that the proposed scheme reduces the runtime overhead significantly.

The rest of this paper is organized as follows. Section 2 provides the brief overview of SRP and the analysis on the runtime overhead of EDF+SRP scheduling. Section 3 explains the proposed scheduling queue structure. In Section 4, the proposed job selection algorithm is verified. The experimental results are presented in Section 5. Finally, Section 6 concludes our work.

2 Analysis on the Selection Overhead

SRP is a real-time synchronization protocol which introduces the concept of preemption level. A preemption level $\pi(J)$ is a static value associated with a job J. The essential property of preemption levels is that a job J_i is allowed to preempt job J_j only if $\pi(J_i)$ is higher than $\pi(J_j)$. In EDF scheduling, for example, the preemption level of a job can be assigned based on the relative deadline of the job; the preemption level is defined such that $\pi(J_i) > \pi(J_j) \Leftrightarrow D_i < D_j$, where D_k is the relative deadline of a job J_k. A ceiling of a resource is defined as the highest preemption level of jobs that may use the resource. The system ceiling is defined as the highest ceiling among the resources that are being used. SRP requires that a job be allowed to start only if its preemption level is higher than the system ceiling. Then, it is guaranteed that the job is not blocked after it starts and there is no deadlock.

An important feature of SRP is the early blocking. The scheduler delays the start of the execution of a job if the job needs a shared resource which is held by another

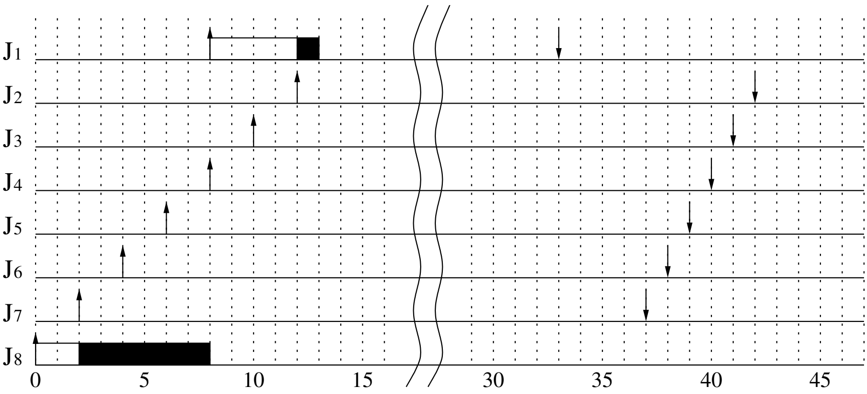
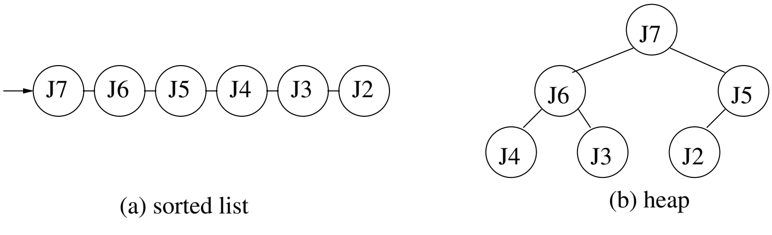


Fig. 1. An EDF schedule of the jobs in Table 1. The arrival time of each job is marked with an upward arrow and the absolute deadline with a downward arrow. Critical sections are represented by colored squares.

Table 1. An example set of jobs

Job #	Arrival time	Relative deadline	Preemption level	WCET	Length of critical section	resource #
J_1	8	25	8	5	1	R_1
J_2	12	30	7	2	1	R_3
J_3	10	31	6	3	1	R_2
J_4	8	32	5	2	1	R_3
J_5	6	33	4	3	1	R_3
J_6	4	34	3	2	1	R_3
J_7	2	35	2	3	1	R_3
J_8	0	100	1	10	6	R_2



(a) sorted list

(b) heap

Fig. 2. The conventional scheduling queue structures

job. After the resource becomes available, once the job starts its execution it will not be blocked. Thanks to this property, the mechanism for accessing shared resources is simple compared with PCP because there is no need to adjust priorities or to block a job. Furthermore, there is no context switch caused by blocking.

In dynamic priority scheduling, the conventional job selection algorithms can find the most eligible job in $O(n)$ time where n is the number of jobs in the system. Figure 1 depicts an EDF schedule of the jobs in Table 1. J_8 arrives at time 0. It acquires R_2 and enters the critical section at time 2. At this time the system ceiling is raised to the resource ceiling of R_2, 6. J_4, J_5, J_6, and J_7 arrive during the critical section of J_8 but they are not allowed to start execution because their preemption levels are not higher than the system ceiling. At time 8, J_1 arrives, and it preempts J_8 since its preemption level is 8. At time 12, J_1 acquires R_1 and enters the critical section. The system ceiling is raised to 8. At time 13, J_1 releases R_1 and completes its execution. The system ceiling is restored to 6. Then, the scheduler tries to find the most eligible job; the preemption level is higher than 6 and the priority is highest among such jobs. If it could not find such a job, it resumes the execution of J_8.

In EDF scheduling, active jobs are kept in a scheduling queue (or ready queue) in the order of absolute deadline. In general, a scheduling queue is implemented using a list or a heap. Figure 2 shows the state of the scheduling queue at time 13 in the above example (J_8 is assumed to have been moved to a *preemption stack.* see [5]). In the list implementation (Fig. 2(a)) a linear search algorithm sequentially examines jobs in the list from the head until it finds a job with a preemption level higher than 6. At last, it finds such a job, J_2, at the tail. In the heap implementation (Fig. 2(b)) the scheduler examines J_7 but it finds that J_7's preemption level is not higher than 6. Then, it examines J_6 and J_5. Although the deadline of J_6 is earlier than J_5, the scheduler also examines J_4 and J_3 because J_6's preemption level is not higher than 6. In this way the scheduler examines all jobs in the scheduling queue, traversing the whole tree. Therefore, using the conventional job selection algorithms and scheduling queue structures, the most eligible job can be found in $O(n)$ time in the worst case.

3 Data Structure for Scheduling Queue

The algorithm presented here uses a selection tree as a scheduling queue structure. For given n jobs, a selection tree with $m = 2^{\lceil \log_2 n \rceil}$ leaf nodes is used. Figure 3 shows an example state for 8 jobs of the example in the previous section. As shown in Fig. 3, the nodes are denoted by from N_1 to N_{2m-1}. Node N_k contains a pointer to Task Control Block (TCB) of the job it represents, denoted by $TCB(N_k)$. The deadline of N_k, denoted by $D(N_k)$, is the absolute deadline of the job that N_k represents. When a job with preemption level p arrives, the job is represented by N_{m+p-1}. An internal node represents the earlier-deadline job of the ones its two children represent. Hence, the root node N_1 represents the earliest-deadline job among all active jobs. If there is no active job at preemption level p, N_{m+p-1} is said to be empty. An internal node, both children of which are empty, is also said to be empty. The deadline of an empty node is defined as ∞. In Fig 3, the number in a node is the absolute deadline of the job the node represents.

The number of preemption level the system supports should be equal to or greater than n which is the number of jobs. Since the selection tree is a complete binary tree, $m = 2^k(2^{k-1} < n \leq 2^k)$. Since $k = \lceil log_2 n \rceil$, $log_2 m = log_2 2^{\lceil log_2 m \rceil} = \lceil log_2 n \rceil$. Therefore, the height of the selection tree is $log_2 m + 1 = \lceil log_2 n \rceil + 1$ and the time complexity of insertion and deletion is $O(\lceil log_2 n \rceil)$.

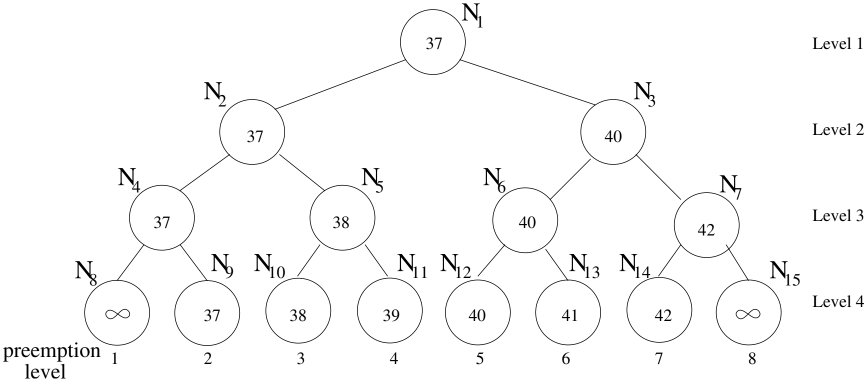


Fig. 3. Selection tree as the scheduling queue structure

It is assumed that each job has a distinct preemption level. If more than one job has the same relative deadline, they are assigned the same preemption level. To accommodate more than one jobs at a single preemption level, the Constant Ratio Grid Method can be applied [7]. A FIFO queue is dedicated to each preemption level. When a job with preemption level p arrives, it is inserted into the p^{th} FIFO queue. The job at the head of the p^{th} FIFO queue is represented by N_{m+p-1}.

4 Selection Algorithm

We define some notations below. $FLAG(N_k)$ tells whether N_k is a left child or a right child. $TREE(N_k)$ is a set of all nodes in the sub-tree rooted from N_k. $LEAF(N_k)$ is a set of leaf nodes in the sub-tree rooted form N_k. $PL(N_k)$ is the preemption level of N_k. Note that the parent node of N_k is $N_{\lfloor k/2 \rfloor}$ and its children are N_{2k} and N_{2k+1} [8].

1. $FLAG(N_k) = \begin{cases} ROOT & : \ k = 1 \\ LEFT & : \ k \neq 1 \quad and \quad k \ mod \ 2 = 0 \\ RIGHT & : \ k \neq 1 \quad and \quad k \ mod \ 2 = 1 \end{cases}$
2. $TREE(N_k) = \begin{cases} \phi & : \ k \geq 2m \\ \{N_k\} \cup TREE(N_{2k}) \cup TREE(N_{2k+1}) & : \ k < 2m \end{cases}$
3. $LEAF(N_k) = \begin{cases} \{N_k\} & : \ m \leq k < 2m \\ LEAF(N_{2k}) \cup LEAF(N_{2k+1}) & : \ k < m \end{cases}$
4. $PL(N_k) = k - m + 1, \quad for \quad m \leq k < 2m$

The selection algorithm is formally described in Algorithm 1. It searches for the most eligible job, *i.e.*, a job whose preemption level is higher than S and the absolute deadline is earliest among such jobs. The search begins from a one-height sub-tree. The most eligible job in the sub-tree is identified as the height of the sub-tree is increases gradually. The sub-tree is denoted by T_k, where N_i is the root node of T_i. The most eligible node in T_k is denoted by N_e.

Initially, both N_k and N_e are made to refer to the leaf node associated with preemption level $S+1$ ($PL(N_{m+S}) = (m+S) - m + 1 = S + 1$). Since the preemption levels are static, the node can be found in $O(1)$. In case that N_k is the left child of its parent and the deadline of its sibling, N_{k+1}, is earlier than the deadline of N_e, N_{k+1} is the most eligible node in the sub-tree $T_{\lfloor k/2 \rfloor}$ which is the sub-tree rooted from the parent node of N_k and N_{k+1}. Note that the preemption level of every node in T_{k+1} is higher than S and the deadline of N_{k+1} is earlier than any other node in T_{k+1}. In case that N_k is the right child of its parent, there is no need to consider N_k's sibling, N_{k-1}, because the preemption level of any node in T_{k-1} is lower than S.

It can be easily shown that the time complexity of Algorithm 1 is $O(\lceil \log_2 n \rceil)$ because the height of the selection tree is $\lceil log_2 n \rceil + 1$. The correctness of the algorithm is shown by Theorem 1.

Lemma 1. *For* $1 \le k < 2m$*, Suppose i is an integer such that* $\log_2(m/k) \le i < \log_2(m/k) + 1$*. Then,*

$$LEAF(N_k) = \{N_p : k \cdot 2^i \le p < (k+1) \cdot 2^i\} \tag{1}$$

Proof: Let T_{N_k} be the sub-tree rooted from N_k. Since T_{N_k} is also a complete binary tree and the level of N_k is $\lfloor \log_2 k \rfloor$, the height of T_{N_k} is $h = \log_2 m - \lfloor \log_2 k \rfloor$ and the number of leaf nodes of T_{N_k} is $2^h = 2^{\log_2 m - \lfloor \log_2 k \rfloor}$. Then, since the leftmost leaf node of T_{N_k} is $N_{k \cdot 2^h}$, $LEAF(N_k) = \{N_k : k \cdot 2^h \le p < k \cdot 2^h + 2^h = (k+1) \cdot 2^h\}$. Since $\lfloor \log_2 k \rfloor = \log_2 m - h = \log_2(m/2^h)$, $\log_2(m/2^h) \le \log_2 k < \log_2(m/2^h) + 1$. Hence, $m \le k \cdot 2^h < 2m$. ■

Corollary 1. *For a node N_k, the highest preemption level of nodes in the tree rooted from the left child of N_k is lower than the lowest preemption level of the nodes in the tree rooted from the right child of N_k.*

Theorem 1. (Correctness) *Algorithm 1 finds the most eligible node.*

Proof: We prove the theorem by induction. Let e_i and k_i be, respectively, e and k after the i^{th} iteration. Then, $k_0 = e_0 = m + S$, $k_1 = \lfloor k_0/2 \rfloor$, $k_2 = \lfloor k_1/2 \rfloor, \ldots, k_m = 1$. Note that $TREE(N_{k_0}) \subset TREE(N_{k_1}) \subset TREE(N_{k_2}) \subset \cdots \subset TREE(N_{k_m})$.

Since $N_{e_0} (= N_{k_0})$ is the unique element in $TREE(N_{k_0})$ and $PL(N_{e_0}) = S + 1$, N_{e_0} refers to the most eligible node in $TREE(N_{k_0})$. Assume that N_{e_i} refers to the most eligible node in $TREE(N_{k_i})$.

Case 1: N_{k_i} is the right child of its parent. Since $N_{k_0} \in TREE(N_{k_i})$ and $PL(N_{k_0}) = S + 1$, the preemption levels of the nodes in $TREE(N_{k_i - 1})$ are not higher than S by

Algorithm 1 Find the earliest-deadline job with $\pi(J) > S$

```
$e \leftarrow m + S$, $k \leftarrow m + S$
repeat
        if $FLAG(N_k) = LEFT$ and $D(N_{k+1}) < D(N_e)$ then
                $e \leftarrow k + 1$
        endif
        $k \leftarrow \lfloor k/2 \rfloor$
until $FLAG(N_k) = ROOT$
return $TCB(N_e)$
```

Corollary 1. None of the nodes in $TREE(N_{k_i-1})$ is eligible. Therefore, $N_{e_{i+1}}$ refers to the most eligible node in $\{N_{\lfloor k_i/2 \rfloor}\} \cup TREE(N_{k_i-1}) \cup TREE(N_{k_i}) = TREE(N_{\lfloor k_i/2 \rfloor}) = TREE(N_{k_{i+1}})$.

Case 2: N_{k_i} is the left child of its parent. Since $N_{k_0} \in TREE(N_{k_i})$ and $PL(N_{k_0}) = S+1$, the preemption levels of the nodes in $TREE(N_{k_i+1})$ are higher than $S+1$ by Corollary 1. All nodes in $TREE(N_{k_i+1})$ are eligible. Since N_{k_i+1} refers to the earliest-deadline node in $TREE(N_{k_i+1})$ by definition, the earlier-deadline node between N_{e_i} and N_{k_i+1} is the most eligible node in $\{N_{\lfloor k_i/2 \rfloor}\} \cup TREE(N_{k_i}) \cup TREE(N_{k_i+1}) = TREE(N_{\lfloor k_i/2 \rfloor}) = TREE(N_{k_{i+1}})$. ■

5 Experiments

We measured the runtime overhead of the scheduler on a GSM/GPRS mobile phone being developed by LG Electronics Inc. The mobile phone is equipped with an ARM7 TDMI processor. The ARM processor used in the experiment does not have I-Cache nor D-Cache, thus we do not need to consider the effect of consistency problems such as cache misses. The target system operates at 52MHz clock speed. To speed up the system, the operating system's codes are loaded from NOR flash ROM into the internal SRAM of the ARM processor during the system is booted up. Also, the operating system is coded using 32-bit ARM instruction set.

The main activities of schedulers are inserting, selecting and deleting a job into/from the ready queue. The associated overheads are called *enqueue overhead* (Δ_{enq}), *selection overhead* (Δ_{sel}), and *dequeue overhead* (Δ_{deq}), respectively. The scheduler is

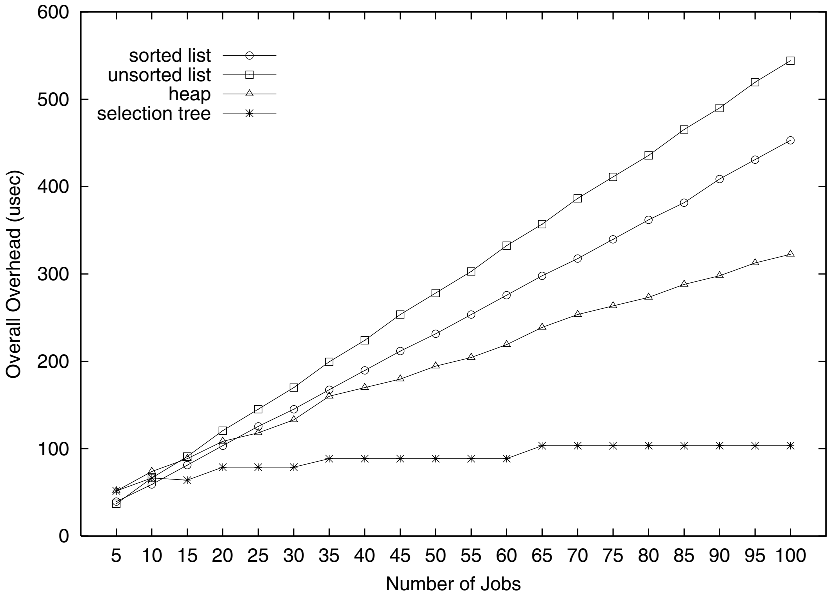


Fig. 4. Per-period overall overhead

invoked to select the most eligible job whenever a new job arrives or a job finishes. Hence, the *overall overhead* per-period is $\Delta t = \Delta_{enq} + \Delta_{dep} + 2\Delta_{sel}$ [9], assuming that the overhead of other parts of operating system is negligible.

We measured the worst case runtime overhead of each activities on the mobile phone. The considered data structures for the scheduling queue are sorted list, unsorted list, heap, and selection tree. The sorted list and the unsorted list was implemented using a doubly linked list. The heap and the selection tree was implemented using an array.

Figure 4 shows the per-period overall overhead (Δt) measured in the experiment. For less than 15 jobs, the proposed algorithm is comparable to the conventional algorithms. For more than 15 jobs, the proposed algorithm reduces the overall overhead significantly owing to the logarithmic time complexity.

The measured data are analyzed and functionalized using the Least Square Method. Table 2 summarizes the measured worst case runtime overhead of scheduler activities. We can see the selection overhead of the unsorted list is greater than that of the sorted list since both the priorities and the preemption levels are compared in the unsorted list, while only the preemption levels are compared in the sorted list to find the most eligible job. As expected, the coefficients of the enqueue and dequeue overhead of the heap are greater than those of the selection tree. This is because the insertion and deletion operation of the heap involves more comparison instruction and swapping a parent node and a child node. As for the selection overhead, the time complexity of the proposed algorithm is $O(\lceil \log_2 n \rceil)$, while those of the conventional algorithms are $O(n)$.

Table 2. Runtime overhead. Values are in μs, and n is the number of jobs

	the conventional algorithms			the proposed algorithm
	sorted list	unsorted list	heap	selection tree
Δ_{enq}	$1.35n + 7.40$	7.39	$4.92\lfloor \log_2 n \rfloor + 2.46$	$3.17\lceil \log_2 n \rceil + 8.75$
Δ_{deq}	4.92	4.92	$6.09\lfloor \log_2 n \rfloor + 2.68$	$3.35\lceil \log_2 n \rceil + 7.08$
Δ_{sel}	$1.42n + 1.64$	$2.63n + 0.15$	$1.17n + 5.39$	$1.94\lceil \log_2 n \rceil + 4.80$

In a today's advanced GSM/GPRS handset supporting many applications such as JAVA, WAP, MP3 player, and high-resolution camera, the number of tasks is usually ranged from 70 to 100. The GSM/GPRS handset used in the experiment has 92 tasks, and up to 74 tasks can be in the ready state simultaneously in the worst case. Thanks to the proposed algorithm, we could reduce the overhead of the scheduler significantly, by about 68.8%. The legacy scheduler that uses a sorted list for 256 priority levels takes about 330 μs for selecting and dispatching a task in the worst case. The proposed algorithm performs the same operations in only about 103 μs. Since the scheduler is activated on every timer interrupt, the shortest period of scheduler activation is about 4.8 *ms*, which is close to a TDMA period. Thus the proposed algorithm consumes about 2.19% of processor utilization, while the legacy with sorted lists consumes about 7.02% of processor utilization in the worst case. Using the proposed algorithm, we can further increase the applications' utilization of the processor.

6 Conclusion

In spite of many strong points, the Stack Resource Policy (SRP) may introduce a considerable amount of scheduling overhead to legacy schedulers because SRP requires that a scheduler should select a job based on both priority and preemption level. In the worst case, the conventional schedulers using EDF and SRP has $O(n)$ scheduling overhead when there are n jobs in the system. This paper presents an algorithm that can find the most eligible job in $O(\lceil \log_2 n \rceil)$ using a selection tree as a scheduling queue structure where jobs are arranged in the order of both priority and preemption level. The correctness of the algorithm is also shown in this paper. Experiments were performed to verify the effectiveness of the proposed scheduling scheme on a GSM/GPRS handset embedding ARM7 TDMI processor. The experimental results show that the algorithm reduces the overall scheduling overhead significantly. The proposed method allows us to further utilize the processor by reducing the worst-case scheduling overhead by about 68.8%.

References

1. Sha, L., Rajkumar, R., Lehoczky, J.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers **39** (1990) 1175–1185
2. Chen, M.I., Lin, K.J.: Dynamic priority ceilings: A concurrency control protocal for real-time systems. Real-Time Systems **2** (1990) 325–346
3. Liu, C., Layland, J.: Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of ACM **20** (1973) 46–61
4. Jeffay, K.: Scheduling sporadic tasks with shared resources in hard-real-time systems. In: Proceedings of 13th IEEE Real-Time Systems Symposium. (1992) 89–99
5. Baker, T.: Stack-based scheduling of real-time processes. Real-Time Systems **3** (1991) 67–100
6. Han, S., Park, M., Cho, Y.: An efficient job selection scheme in real-time scheduling under the stack resource policy. In: Proceedings of the 17th International Parallel and Distributed Processing Symposium. (2003) 118
7. Liu, J.W.S.: Real-Time Systems. Prentice-Hall (2000)
8. Knuth, D.E.: The Art of Computer Programming. Volume 3. Addison-Wesley (1998)
9. Zuberi, K.M., Pillai, P., Shin, K.G.: EMERALDS: a small-memory real-time microkernel. In: Proceedings of 17th ACM Symposium on Operating Systems Principles. (1999) 277–299

A Programming Model for an Embedded Media Processing Architecture*

Dan Zhang[1], Zeng-Zhi Li[1], Hong Song[2], and Long Liu[1]

[1] School of Electronics & Information Engineering,
Xi'an Jiaotong University, Xi'an Shaanxi 710049, China
danzhang@mailst.xjtu.edu.cn

[2] School of Mechanical Engineering, Xi'an Shiyou University,
Xi'an Shaanxi 710065, China

Abstract. To follow rapid evolution of media processing algorithms, the latest media processing architecture enhances the execution efficiencies of media applications by adding a programmable vision processor and by improving memory hierarchy, while complicates the programming. In this paper, the features of this architecture are analyzed, the reason of inefficiency of media application implemented by general programming model is studied and SPUR programming model is proposed. In SPUR, media data and operations are expressed as media streams and corresponding operations naturally. Moreover, algorithm is divided into high-level part written by SP-C and low-level part written by UR-C. Fine-grained data parallelism are exploited explicitly as well. Experimental results show that SPUR provides programmer a novel, expressive and efficient programming way, and obviously improves readability, robustness, development efficiency and object-code quality of media applications.

1 Introduction

There is a wide range of applications of media processing to capture, store, manipulate and transmit media objects such as text, handwritten data, audio objects, still images, 2D/3D graphics, animation and full-motion video. Each media processing environment requires different processes, techniques, algorithms and hardware [1], [2], [3]. A variety of algorithms have come into series of standards, such as MPEG-1, MPEG-2, MPEG-4, JPEG2000, H.263 and H.264. At present, computer applications are becoming media-rich and WWW will make future applications contain more and more media objects [4], [5], [6], which present great challenges to both hardware and software environment of media processing.

The key characteristics of media applications include high computing rate, high data bandwidth, low global data reuse and high data locality, ample parallelism at the instruction, data and task levels, and real-time requirements [7], [8]. Many researches have been made on the architecture and programming technologies of media processing for a long time [3], [9], [10], [11], [12]. The differences of application environments lead to

* This work has been supported by the Nation Science Foundation of China (No.60173059).

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 251–261, 2005.

various requirements of performance, area, power and cost. Thus, media processing systems have to make trade-off between hard-wire and software programmability. Media Processor (MP), a kind of programmable media-specific processor, makes such trade-off successfully and gains higher performance than General-Purpose Processor (GPP) and PDSP. Moreover, a novel Media Processor with Programmable Vision Coprocessor (MPwPVC) introduces master-slave mode to executes scalar and vector operations in parallel, and three-tier memory hierarchy to increase available bandwidth. MPwPVC improves performance, but complicates the programming. General programming model (GPM) is not suitable for this complexity because of its emphasis on expressiveness and flexibility of language and wide range of applications. Hence, it is significant to propose a programming model for MPwPVC.

In this paper, the features of this architecture are analyzed, the reason of inefficiency of media application implemented by GPM is pointed out, and SPUR programming model is proposed. In SPUR, media data are expressed as media streams naturally, and the general code of media application are reconstructed into high-level skeleton programs and low-level microcode routines, which are compiled, assembled and linked into object code respectively. SP-C, used to write skeleton programs, simply extends ANSI C. UR-C, used to write microcode routines, treats media streams as kernel data objects, expresses the inherent data parallelism in media algorithms explicitly, and extends operations on media streams. The experimental results show that the development efficiency and object code quality of media applications on MPwPVC are improved obviously by applying SPUR programming model. This paper is organized as follows. The comparisons with related studies are presented in section 2. The overview of MPwPVC is presented in section 3. SPUR programming model is proposed in section 4. The experimental results are provided in section 5. Finally, the conclusions and future works are given in section 6.

2 Comparisons with Related Studies

Vector C [13] targets general vector machines. The arrays are treated as first-class objects by using a special subscripting syntax to select array slices. Many new vector operators are added, but no ones specific to media processing. Similarly C[](C brackets) [14] also introduces lots of vector operators. Specific to the GPPs with media ISA extensions, MMC [15] extends syntax and semantic to support vector data type and necessary vector operators. The Compilers of TI [16] and Intel [17] provide intrinsic libraries for media ISA extension. They, however, are only used to develop media software for some particular class of processors (e.g. Intel family). DSP-C [18] extends ANSI C with fixed point type, saturation type, circular buffers type and multi-memory type to support the most common characteristics of DSPs explicitly and generate object-code of higher quality. SWARC [19] is a portable, high-level SWAR execution model, supporting GPP with SIMD extensions. It presents SWAR layout in high-level language (HLL) and adds some vector operators. In Trimedia [20], VLIW RISC performs most media processing tasks with SIMD, while coprocessors do fixed tasks. SIMD operations are exposed as custom operations. Trimedia Stream Software Architecture (TSSA) emphasizes modulization, interoperability, reusability and compatibility. Vision Instruction Processor

(VIP) [21] has PE-array and special cache system designed for vector and matrix type operations. The media kernels are developed by using VPL and compiled into assembly code of microcontroller and OAK DSP C code. The high-level programs are developed by using C++. Imagine [22] is a stream architecture and programming model, which contains a stream-specific memory hierarchy and supports a large quantity of processing elements, then improves performance and memory bandwidth. To overcome the shortcomings of mismatch with application-domain and unreasonable abstraction of general-purpose languages on stream processing grid-based architecture, StreamIt [23] language provides high-level structured abstract representations of stream and a messaging system for control on grid-based architecture.

Compared with above studies, UR-C refers some vector operators from C[] and Vector C and MMC to apply to media streams. The concept of media stream comes from Imagine and StreamIt, and thought of algorithm partition derives from VIP and Imagine. Differently SPUR is specific to MPwPVC. TSSA is only used to develop high-level part of algorithm, VPL to develop low-level part. Unlike them, SPUR provides full programmability. Saturation operations in DSP-C act implicitly as type qualifiers, but UR-C adds explicit saturation operators to enhance safety and reliability. In a word, some features of SPUR come from other studies, but its design goals and key decisions are not same.

3 Overview of MPwPVC Architecture

In general MP, RISC controls other components and implements media algorithms by software programming, while coprocessors deal with various urgent, real-time, high-throughput and fixed tasks. The uni-processor cannot perform scalar and vector operations efficiently at same time, and memory system also cannot provide high-speed access of two kinds of data. Therefore, as fig. 1 shows, a novel MPwPVC architecture contains following features:

1. High performance parallel processing ability: The computations are distributed among various parts of MPwPVC. In VLIW and SIMD ways, PVC performs a lot of vector operations for vision processing, which is most computation-intensive, change-frequent and algorithm-evolution-rapid. RISC only performs high-level part of algorithms and scalar operations.
2. Separated scalar and vector data: According to their different processing and accessing ways, scalar data are stored in SRAM and vector data in DRAM. SRAM Bus connects scalar function units and DRAM Bus connects vector ones. This reduces conflict-probability of access, so increases bandwidth.
3. Three-tier memory hierarchy of vector data: It consists of off-chip DRAM, on-chip VRAM and RF, which is suitable for the storing and accessing pattern of media data very well, and exploits the features of high data locality, low global data reuse and concurrency in media applications efficiently.

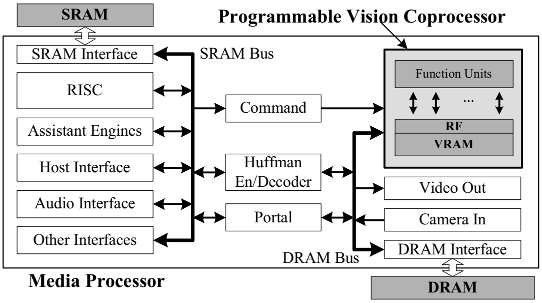


Fig. 1. Block diagram of Media Processor with Programmable Vision Coprocessor

4 SPUR Programming Model

MPwPVC improves the performance of media applications, but complicates software programming. The high development-cost of assembly and unfitness of GPM are the reasons of new programming model. The design principles of SPUR are to decrease the application range of language and model, heighten the abstraction level, enhance the ability of expression of media application, and map well to architectural features of MPwPVC. The previous languages and models provide abstractions of underlying hardware of GPP or vector processor and have a large range of application, so they have great flexibilities. However, SPUR provides a high-level abstraction of media applications, makes programmer focus more attention on implementation of algorithms than hardware and Instruction Set Architecture(ISA) details. Meanwhile, it also has good mappings of architectural features of MPwPVC, then simplify the works of compiler.

4.1 Design Goals

The model design was guided by a handful of high-level goals:

Ease of programming: The media applications and algorithms have evolving rapidly, so new product usually is time-crucial to market. But assembly language implies a lot of well-known drawbacks such that the development is slow and painful. GPM, which emphasizes expressiveness and flexibility of language and wide range of applications, is not suitable for this complexity.

Performance: MPwPVC must provide real-time high performance for media processing, thus the compiler ability is a key consideration in model design.

Portability: The model must support porting applications between different MPwPVCs. The hardware details are hidden in HLL and compiler is responsible for porting to target machine.

Complete support for hardware functionalities: Programmers will be hard to depend on HLL and programming model if it blocked access to functionalities that are available in assembly language.

Good interface to assembly library: If new model do not have it, the rate of code reuse is low and acceptance of new model by programmer is slow.

Extensibility for future hardware: The model and language should be extended to fit new features without breaking backward compatibility.

4.2 Introduction of SPUR Programming Model

The main definitions of SPUR programming model are listed below:

Media Stream: A sequence of data record with a regular pattern. It is a algorithm-level object and independent of memory layout.

Microcode Routine: A sequence of operations performing on a media stream. It is used to implement the low-level part of algorithms and also called kernel [22]. Usually it reads input streams, performs computing, and then writes output streams. UR-C (Ucode Routine C) is used to write it.

Skeleton Program: It is used to define streams and implement high-level part of algorithms. It exposes the control flows and data flows among microcode routines. SP-C (Skeleton Program C) is used to write it.

SPUR Programming Model: A programming method to develop media application on MPwPVC architecture, which includes SP-C and UR-C languages, and corresponding compilers, assemblers, linkers and simulators.

In SPUR model, general-purpose code of media application are required to be reconstructed as following steps:

1. Partition algorithm: Analyze algorithm and partition it into low-level and high-level parts, which correspond respectively to higher syntax layer (e.g. Picture and GOB) and lower syntax layer (e.g. MB, Block and pixel).
2. Extract media streams: In general-purpose code, an array operated in loop usually represents a Block, MB or GOB. It can be transformed into media stream or other derived streams of it according to their access patterns.
3. Construct microcode routines: The regular data access patterns usually exist in loops of low-level parts, which can be transformed into **check_stream** loop in UR-C. Then microcode routines are made with high cohesion and low coupling according to the inherent relations of modules in algorithms.
4. Construct skeleton program: Skeleton program performs high-level control and data flow tasks, and then builds whole application. These tasks include controlling peripherals, inputting/outputting data, controlling other components for assistant tasks, and high-level syntax parsing of media data.

As fig. 2(a) shows, media applications contain data localities, which means a sequence of microcode routines deal with a stream one by one and intermediate results are needless to be written out. As fig. 2(b) shows, the intermediate results are placed in RF or VRAM near function units, therefore greatly reduce the requirements for global bandwidth. The coarse-grained task-level parallelism exists between microcode routines, and fine-grained instruction and data level parallelism exist in microcode routines. Skeleton programs run in MP, while microcode routines run in PVC.

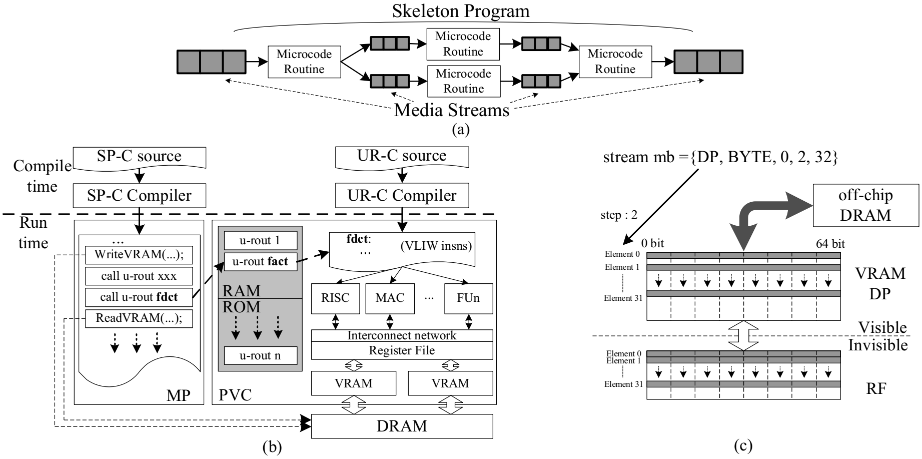


Fig. 2. (a) Media streams processing flow; (b) Runtime environment of SPUR programming model; (c) Runtime memory image of a 16×16 Macro-Block media stream

4.3 UR-C (Ucode Routine C)

UR-C has more limitations than ANSI C, such as not allowing global variables, pointers, etc. It limits some unnecessary flexibilities for media applications, and then improves the efficiency of compiler. The features of UR-C are listed below:

1. Combination data type: It maps media data with 8 (**HALF**), 16 (**BYTE**), 32 (**WORD**), 64 (**DWORD**) and 128 bit width (**QWORD**) explicitly. They indicate how to split SIMD function units to execute operations such as addition, multiply-accumulate and transpose.
2. Media stream data type: Media stream resides in on-chip VRAM and RF. SP-C programs execute the transfers between DRAM and VRAM and UR-C programs execute various SIMD operations on media streams in VRAM. RF is invisible to programmers, so the instructions, which transfer data between VRAM and RF and operate on RF, are generated by compiler. By this way, programmers express data flows and high-level control flows explicitly, while low-level instructions are generated by compiler. Programmer and compiler manipulate media streams through Media Stream References (MSRs).

$$MSR := < B, M, O, S, N, C > \tag{1}$$

B indicates memory space, *M* indicates combination type, *O* indicates offset, *S* indicates step, and *N* indicates the number of word. These are all user-properties. *C* indicates system property "cursor" which is updated by instructions generated by compiler. MSR can be defined recursively, namely derived-streams. The element of media streams must be a full machine word. The syntax of MSR is defined as follows:

msr: 'stream' IDENTIFIER '=' ms_expr
ms_expr: ms_expr_basic | ms_expr_derived
ms_expr_basic: '{' MSPACE ',' CTYPE ',' OFF ',' STEP ',' ELENUM '}'
ms_expr_derived: 'derived_stream(' msr ',' CTYPE ',' OFF ',' STEP ',' ELENUM ')'

The primitives operating on media streams are defined as follows:

stream_ref.read(): Reads the element pointed by cursor;
stream_ref.write(val): Writes val into the element pointed by cursor;
stream_ref.step(): Advances the cursor by one element;
stream_ref.pop(): Associates read and step;
stream_ref.push(val): Associates write and step;

Figure 2(c) shows the runtime memory image of a media stream defined on a processor with 64-bit width word, representing a 16×16 Macro-Block.

3. Structured data access: UR-C only allows MSR and scalar variables to be accessed through parameters, not allows global variables.
4. Limited control flow: Because all operations are based on media streams, general **for**, **while** and **do-while** loops are removed, and loop construct **check_stream**(*msr*) is added. It checks whether cursor of *msr* reach the end. It simplifies program structure, decrease control dependencies and make compiler generate higher quality code.
5. Saturation operators: UR-C supports saturation operation explicitly. As the following example shows, the sum result of stream *a* and stream *b* will be saturated according to 16 bit **HALF** independently.

```
stream a = {DP, HALF, 0, 1, 32}, b = {DP, HALF, 32, 1,32},
   c = {DM, HALF, 0, 1, 32};
c = addsat(a, b);
```

6. Reduction operators: Unary reduction operation means executing binary operation on media stream element by element and getting a scalar result. In the following example, all elements of *sa* are summed into *res*. All reduction operators are shown in table below.

```
stream sa = {DP, HALF, 0, 1, 32};
res = @+ sa;
```

Operator	@+	@*	@<	@>	@|	@&	@ˆ
Description	Sum	Product	Minimum	Maximum	Logic OR	Logic AND	Logic XOR

7. Segment operator: It is used to get sub-word of a media stream. In the following example, the first **BYTE** of *a* is assigned to *temp*.

```
temp = segment(a, BYTE, 0);
```

4.4 SP-C (Skeleton Program C)

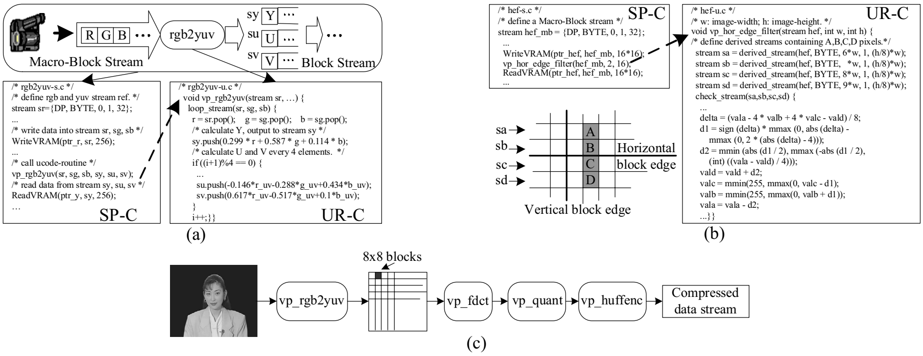
SP-C is the superset of ANSI C and has following simple extensions:

1. Microcode routine library: All prototypes of callable microcode routines are defined in *u-rout.h*. SP-C programs call microcode routines like functions. In the following syntax, arguments can be with MSR type or any scalar type.
   ```
   ucode_name(argument 0, ..., argument n);
   ```
2. Manipulation functions of media stream: Media streams reside in VRAM and RF, and whose data are from off-chip DRAM. As fig. 2(b) shows, SP-C includes some functions of media stream controlling the data-transferring between DRAM and VRAM. The syntax is defined as follows.
   ```
   WriteVRAM(char *ptr, stream s, int byte_cnt);
   ReadVRAM(char *ptr, stream s, int byte_cnt);
   ```
3. APIs accessing other components: Except calling microcode routines to perform vision processing tasks, SP-C programs also need controlling other components to finish some works, e.g. protocol parsing and variable length encoding/decoding, etc.

5 Experimental Results

To evaluate the availability and efficiency of SPUR model, VFAST, a typical MPwPVC, is selected as target machine [24]. UR-C compiler makes use of soft-pipeline and other loop-optimization, and scheduling technique of reading and writing of media streams. Three media application examples are as follows.

1. RGB-to-YUV [25]: As fig. 3(a) shows, SP-C program captures the raw image from camera, parses it into 16×16 pixel RGB Macro-Block streams *sr*, *sg*, *sb* according to syntax, then calls microcode routine *vp_rgb2yuv* to compute and generate six YUV Blocks (4:1:1), finally outputs *sy*, *su*, *sv* Block streams.



```
/* rgb2yuv-s.c */
/* define rgb and yuv stream ref. */
stream sr={DP, BYTE, 0, 1, 32};
...
/* write data into stream sr, sg, sb */
WriteVRAM(ptr_r, sr, 256);
...
/* call ucode-routine */
vp_rgb2yuv(sr, sg, sb, sy, su, sv);
/* read data from stream sy, su, sv */
ReadVRAM(ptr_y, sy, 256);
…
```

```
/* rgb2yuv-u.c */
 void vp_rgb2yuv(stream sr, …) {
  loop_stream(sr, sg, sb) {
   r = sr.pop();    g = sg.pop();    b = sg.pop();
   /* calculate Y, output to stream sy */
   sy.push(0.299 * r + 0.587 * g + 0.114 * b);
   /* calculate U and V every 4 elements.  */
   if ((i+1)%4 == 0) {
    ...
    su.push(-0.146*r_uv-0.288*g_uv+0.434*b_uv);
    sv.push(0.617*r_uv-0.517*g_uv+0.1*b_uv);
   }
   i++;}}
```

```
/* hef-s.c */
/* define a Macro-Block stream */
stream hef_mb = {DP, BYTE, 0, 1, 32};
  ...
  WriteVRAM(ptr_hef, hef_mb, 16*16);
  vp_hor_edge_filter(hef_mb, 2, 16);
  ReadVRAM(ptr_hef, hef_mb, 16*16);
  ...
```

```
/* hef-u.c */
/* w: image-width; h: image-height. */
void vp_hor_edge_filter(stream hef, int w, int h) {
/* define derived streams containing A,B,C,D pixels.*/
 stream sa = derived_stream(hef, BYTE, 6*w, 1, (h/8)*w);
 stream sb = derived_stream(hef, BYTE,   *w, 1, (h/8)*w);
 stream sc = derived_stream(hef, BYTE, 8*w, 1, (h/8)*w);
 stream sd = derived_stream(hef, BYTE, 9*w, 1, (h/8)*w);
 check_stream(sa,sb,sc,sd) {
  ...
  delta = (vala - 4 * valb + 4 * valc - vald) / 8;
  d1 = sign (delta) * mmax (0, abs (delta) -
    mmax (0, 2 * (abs (delta) - 4)));
  d2 = mmin (abs (d1 / 2), mmax (-abs (d1 / 2),
    (int) ((vala - vald) / 4)));
  vald = vald + d2;
  valc = mmin(255, mmax(0, valc - d1);
  valb = mmin(255, mmax(0, valb + d1));
  vala = vala - d2;
  ...}}
```

Fig. 3. (a) RGB-to-YUV; (b) Horizontal edge filter; (c) JPEG-DCT encoder

Table 1. The comparison between UR-C and handwritten assembly benchmarks

u-routine	Data size	Description	U(c)	U(s)	A(c)	A(s)	p-rat	s-rat
vp_rgb2yuv	16 × 16	RGB to YUV	847	448	683	344	0.81	1.3
vp_fdct	8 × 8	Forward DCT	1352	888	936	604	0.69	1.47
vp_quant	16 × 16	Quantification	11782	1688	4549	836	0.39	2.02
vp_idct	8 × 8	Inverse DCT	1590	1132	980	592	0.62	1.91
vp_thr_le2thr	8 × 8	Clip below threshold	231	172	137	112	0.59	1.54
vp_hor_edge_filter	16 × 16	Horizontal Edge Filter	1167	700	638	408	0.55	1.72
vp_me	8 × 8	Motion estimation	2013	400	952	196	0.47	2.04
vp_conv_3x3	8 cols	3 × 3 convolution	1163	368	506	192	0.44	1.92
vp_median_3x3	32 cols	Median 3 × 3 filter	1308	1024	734	516	0.56	1.98

2. Horizontal edge filter [26]: As fig. 3(b) shows, in *vp_horiz_edge_filter*, the pixel-sequences A,B,C,D around horizontal edge of blocks are defined as four derived streams sa,sb,sc, and sd of input stream hef.
   In above two examples, each element has **BYTE** combination type, which makes eight pixels be performed in SIMD way. In comparison with general-purpose array-based code, the media-stream-based code avoid the computation of array sub-script, multi-loop control and possibility of array out-of-range, moreover, the expression of data and operation are more natural and direct. The whole program has simplicity, naturalness and robustness.
3. JPEG encoder based on DCT (JPEG-DCT) [27]: The basic steps of JPEG-DCT encoder are shown in fig. 3(c). Media streams representing a still image pass through the microcode routines *vp_rgb2yuv*, *vp_fdct*, *vp_quant* and *vp_huffenc*, finally generates compressed data stream.

Table 1 shows the results comparison between UR-C and handwritten assembly code in terms of cycles and code size. All UR-C kernel programs are compiled by elementary UR-C compiler and measured in cycle-accurate simulator VPSIM. Column 1 shows the name of media kernels. Column 2 shows data size. Column 3 shows description. Column 4 and 5 show the cycles and sizes of UR-C, and column 6 and 7 show the cycles and sizes of handwritten ASM. The last two columns p-ratio and s-ratio indicate the performance ratio (0.39 $\sim$ 0.81) and code size (1.3 $\sim$ 2.04) ratio between UR-C and handwritten assembly. These results are not very satisfying, therefore improvements are needed.

6 Conclusion and Future Works

As a novel embedded programmable media processor, MPwPVC introduces a PVC and memory hierarchy of vector data to satisfy the requirements of performance, area, cost and power, but complicates programming. By using SPUR programming model for MPwPVC, the media data can be expressed naturally, simply and directly, and data-level parallelism can be exploited explicitly. The reconstruction method of media applications is presented, and the corresponding languages and compilers are proposed.

Accordingly the readability, robustness, development efficiency and object-code quality of media applications are improved obviously. However, there are some spaces for improvement in language design and optimization techniques of compiler.

References

1. Lee, R.B., Smith, M.D.: Media processing: A new design target. IEEE Micro (1996) 6–9
2. Lee, R.B.: Accelerating multimedia with enhanced microprocessors. IEEE Micro (1995) 22–32
3. Dasu, A., Panchanathan, S.: A survey of media processing approaches. IEEE Trans. on Circ. and Sys. for Video Tech. **12** (2002) 633–644
4. Furht, B.: Processor architectures for multimedia: A survey. In: Multimedia Modeling Conf. (1997) 89–109
5. Sasaki, H.: Multimedia complex on a chip. In: IEEE Inter. Solid-State Circuits Conf. (1996) 16–19
6. Lev, L.A., et al: A 64-b microprocessor with multimedia support. IEEE Journal of Solid-State Circuits **30** (1995) 1227–1238
7. Owens, J.D., et al: Media processing applications on the imagine stream processor. In: IEEE International Conference on Computer Design. (2002) 295–302
8. Aron, N., et al: Study of multimedia application characteristics. [Online]. http://www. stanford.edu/class/ee392c/handouts/apps/media_long.pdf (2003)
9. Pirsch, P., Stolberg, H.J.: Vlsi implementations of image and video multimedia processing. IEEE Trans. on Circ. and Sys. for Video Tech. **8** (1998) 878–891
10. Panchanathan, S.: Architectural approaches for multimedia processing. In: ACPC'99, LNCS 1557. (1999) 196–210
11. Lappalainen, V., et al: Overview of research efforts on media isa extensions and their usage in video coding. IEEE Trans. on Circ. and Sys. for Video Tech. **12** (2002) 660–670
12. Shahbahrami, A., Juurlink, B., Vassiliadis, S.: A comparison between processor architectures for multimedia applications. In: RISC2004. (2004)
13. Guštin, V., Bulić, P.: Introducing the vector c. In: 5th Inter. Meeting on High Perf. Comp. for Computational Science VECPAR. (2002) 253–266
14. Kalinov, A., et al: An ansi c superset for vector and superscalar computers and its retargetable compiler. Journal of C Language Translation **5** (1994) 183–198
15. Bulić, P., Guštin, V.: An extended ansi c for processors with a multimedia extension. International Journal of Parallel Programming **31** (2003)
16. TI: Tms320c6000 optimizing compiler user's guide (rev. l). [Online]. http://www-s.ti.com/sc/psheets/spru187l/spru187l.pdf (2004)
17. Intel: Intel c++ compiler 8.1 for linux. [Online]. http://www.intel.com/software/products/compilers/clin/ (2005)
18. Beemster, M., van Someren, H.: The dsp-c extension to c. [Online]. http://www.techonline.com/community/tech_group/dsp/tech_paper/36995 (2003)
19. Fisher, R.J., Dietz, H.G.: Compiling for simd within a register. In: 11th Inter. Workshop on Lang. and Comp. for Parallel Computing. (1998) 290–304
20. Philips: Trimedia sde. [Online]. http://www.alacron.com/downloads/vncl98076xz/sde_2_75006255.pdf (2000)
21. Ramacher, U., et al.: A 53-gops programmable vision processor for processing, coding-decoding and synthesizing of images. In: 31st European Solid-State Device Research Conference. (2001)
22. Kapasi, U.J., et al.: Programmable stream processors. IEEE Computer (2003) 54–62

23. Thies, W., Karczmarek, M., Amarasinghe, S.: Streamit: A language for streaming applications. In: Inter. Conf. on Compiler Construction. (2002)
24. Leadtek: Vfast architectural reference manual. Something (2001)
25. Pollard, N., May, D.: Using interval arithmetic to calculate data sizes for compilation to multimedia instruction sets. In: ACM Multimedia ’98. (1998) 279–284
26. Lim, J.S. In: Two-Dimensional Signal and Image Processing. Prentice Hall (1990) 478–488
27. Wallace, G.K.: The jpeg still picture compression standard. Communications of the ACM **34** (1991) 30–44

Automatic ADL-Based Assembler Generation for ASIP Programming Support[1]

Leonardo Taglietti, Jose O. Carlomagno Filho, Daniel C. Casarotto,
Olinto J.V. Furtado, and Luiz C.V. dos Santos

Computer Science Department,
Federal University of Santa Catarina, Florianópolis, SC, Brazil
leonardo@inf.ufsc.br
http://www.inf.ufsc.br

Abstract. Systems-on-Chip (SoCs) may be built upon general purpose CPUs or application-specific instruction-set processors (ASIPs). On the one hand, ASIPs allow a tradeoff between flexibility, performance and energy efficiency. On the other hand, since an ASIP is not a standard component, embedded software code generation cannot rely on pre-existent tools. Each ASIP requires a distinct toolkit. To cope with time-to-market pressure, automatic toolkit generation is required. Architecture description languages (ADLs) are the ideal starting point for such automation. This paper presents robust and efficient techniques to automatically generate a couple of tools (assembler and pre-processor) from the ADL description of a given target processor. Tool robustness results from formal techniques based on context-free grammars. Tool efficiency evidence is provided by experiments targeting three CPUs: MIPS, PowerPC 405 and PIC 16F84.

1 Introduction

As a result of the sheer rate of growth expressed by Moore's Law, millions of gates are available within a single chip. This huge supply of hardware combined with the growing demand from the embedded systems industry gave rise to *Systems-on-Chip* (SoCs)[1]. SoCs may be built upon general purpose CPUs or application-specific instruction set processors (ASIPs). Being tailored to a single application class, an ASIP allows a tradeoff between flexibility, performance and energy consumption [5]. Since an ASIP is not a standard component, embedded software code generation cannot rely on pre-existent tools like compiler, instruction-set simulator, assembler, etc. On the contrary, each ASIP requires its own toolkit. To cope with time-to-market constraints, automatic toolkit generation is required. A starting point for such automation should be some formal description of the target CPU. It would be convenient to adopt a SystemC [3] description as a starting point, as it is considered one of the most promising languages for SoC modeling [3]. However, a model written directly in SystemC would admit so many different description styles that automation would be cumbersome. This problem can be avoided by the use of Architecture Description Languages (ADLs). Among those reported in the

[1] This work is supported by CNPq/Microelectronics National Program (Grant 132930/2003-0).

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 262–268, 2005.

literature (see Section 2), we have chosen ArchC [2], an ADL under the *GNU Lesser General Public License* which allows the automatic generation of SystemC models. The scope of this work is the development of robust and efficient techniques to support automatic toolkit generation. This paper focuses on the automatic generation of a couple of related tools (pre-processor and assembler) from the description of the instruction-set architecture (ISA) of a given ASIP. The remainder of this paper is organized as follows. Section 2 describes related work. Automatic tool generation is detailed in Section 3. Section 4 summarizes experimental results. Conclusions and future work perspectives are presented in Section 5.

2 Related Work

2.1 Automation Evolution

Assembler generation techniques have improved a lot since the late seventies. Early *ad-hoc* techniques simply took advantage of similarities among assembly languages[16]. Later, formal techniques based on grammars were introduced to allow the automatic generation of assembly language parsers [17]. However, grammar generation was performed semi-automatically. In the first attempts to address automatic grammar generation, relevant information was filtered from an HDL-like description of the target CPU and was then cast in an intermediate representation more suitable for grammar generation. More recently, with the advent of ADLs, that cumbersome filtering was replaced by ADL parsing [6] [7][10][11][12].

2.2 Parser Generation Engines

Most assembler generators reported in the literature[6][14][15] use Lex and Yacc for both ADL and assembly language parser generation. Besides its inherent limitation to LALR(1) syntactic analysis[13], Yacc does not have a co-validation mechanism between language and grammar. To avoid such limitations, we adopt GALS[9], a lexical and syntactic analyzer generator developed in our department. It assumes a context-free grammar (CFG) representation of a given language. GALS relies on a unified specification of both lexical and syntactical aspects and allows a broader range of syntactical analysis techniques like recursive top-down, predictive LL(1), SLR(1), LALR(1), and LR(1) [13]. Its most important feature is the availability of an embedded simulator for co-validation between language and grammar, as well as debugging facilities for regular expression verification. This improves the maintainability and extensibility of the assembler generation tool.

2.3 The Scope of Our Contribution

In this paper, we use GALS as the key engine to build an assembler generator for the ADL ArchC [2]. ArchC is licensed under *GNU Lesser General Public License* and has the unique feature of generating SystemC models automatically. The current ArchC package (version 1.2) allows the automatic generation of instruction-set simulator, compiled simulator generator and co-verification tools. Therefore, this work contributes to current ArchC toolkit by adding an assembler generator.

3 Automatic Tool Generation

Our automatic assembler generation flow consists of the following steps, which are detailed in Figure 1.

- ADL parser generation
- ADL parsing
- Instruction list generation
- CFG generation for the assembly language
- Assembly-code parser generation
- Assembler tool compilation

GALS is the key to automatic parser generation and is therefore used in the first and fifth steps. The first step is implemented by feeding GALS with a CFG for the adopted ADL. In the second step, the ADL parser extracts information from the ADL

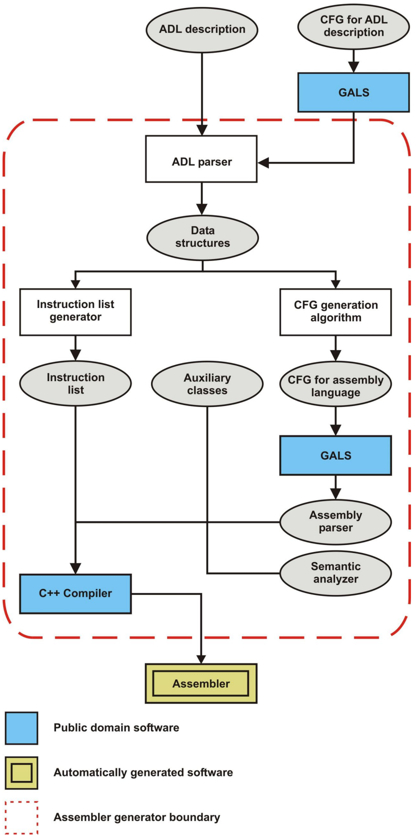


Fig. 1. Assembler generation flow

description and stores them into internal data structures. In a third step, specialized functions look up the data structures and build an instruction list. A CFG generation algorithm was designed to accomplish the fourth step by extracting information from the data structures. The resulting CFG for the assembly language is fed as input to GALS in the fifth step, leading to the assembly-code parser. In the last step, the instruction list, the assembly-code parser, some auxiliary classes and the semantic analyzer are all compiled, resulting in the generation of the desired assembler tool. It should be noted that the generated CFG is LL(1) and thereby unambiguous, which assures deterministic syntactic analysis, granting robustness to the assembler generation tool.

As far as the ADL is frozen, the ADL parser is a standard component of the assembler generator. However, if the ADL has to be changed or extended, a new ADL parser can be efficiently and safely generated by the GALS tool, granting maintainability and extensibility to our assembler generator. Note also that the generator components were deliberately designed so that only the instruction list and the assembly's CFG must be changed during the generation of a new assembler, since only them are dependent on the described instruction-set architecture.

The generated assembler assumes that all assembly instructions are native. Since the use of pseudo-instructions and macros is quite common, a pre-processor should be used beforehand to guarantee that only native instructions are present in the actual assembly code. The role of such pre-processor is to expand macros, replace pseudo-instructions by native ones, calculate target addresses for branches, etc. Essentially, our pre-processor generator extracts information from an input set-up file (containing directives, pseudo-instructions and address generation rules) reuses the instruction list depicted in Figure 1 and adds address generator and standard classes to produce the pre-processor tool.

4 Experiments

4.1 Experimental Set-Up

Our experiments were run on a CPU Intel® Pentium 4, 1.8 GHz, with 256 MB of main memory, under Linux Debian. Although our ultimate goal is ASIP support, for the sake of tool validation and without loss of generality, well-known general purpose ISAs were used for the experiments: MIPS, PowerPC 405 and PIC 16F84. The binary code generated by the assembler tool was executed on functional models of each target CPU, which were generated from ArchC descriptions. The MIPS ArchC model was obtained in [2]. PIC 16F84 and PowerPC 405 models were developed by the authors. Programs extracted from the Dalton Project[4] were used as benchmarks. Execution time is expressed in seconds and was measured by adding CPU *user* and *system* times after invoking the Linux *time* command.

4.2 Experimental Procedures

For each target CPU C_i, represented by a model M_i written in ArchC, the following tools were generated: an instruction-set simulator S_i (generated by the ArchC package itself), a pre-processor P_i and an assembler A_i. Each benchmark program was compiled to the target CPU C_i, resulting in the respective assembly code. Finally, the assembly code of each benchmark was submitted to pre-processor P_i and then to assembler A_i, resulting in the binary code for target CPU C_i.

4.3 Experimental Results

Our experiments aim at validating and checking the efficiency of both the *generating* and the *generated* tools. Results for the generating tools are shown first (Tables 1 and 2); results for the generated tools are shown later (Tables 3 and 4). Table 1 shows the time needed to generate the pre-processor and the assembler tools.

Table 1. Pre-processor and assembler generation times

CPU	ISA size	Pre-processor	Assembler
MIPS	58	0,04	1,13
PowerPC	120	0,05	3,44
PIC	35	0,03	1,06

Clearly, assembler generation time is dominant over pre-processor generation time, which can be neglected. As expected, assembler generation times are essentially proportional to ISA sizes. To investigate the rate of growth with ISA size, different cores were built with growing subsets of a same ISA, as follows: given a core C_i, core C_{i+1} contains all instructions of C_i plus ten new instructions. Table 2 shows that time grows linearly with ISA size.

Table 2. Assembler generation time

Core	ISA size	PowerPC	MIPS
C_1	15	0,97	0,98
C_2	25	1,00	1,01
C_3	35	1,02	1,02
C_4	45	1,03	1,03
C_5	55	1,10	1,10

Table 3 characterizes the benchmarks used in the experiments with the generated tools. For a given ISA, the left column represents the number of executed instructions in each benchmark and the right column (bold) represents the code size (number of instructions in the code).

For the sake of validation, the binary code of each benchmark was run on every target CPU. The experimental results were consistent for all tested cases. For the purpose of efficiency evaluation, the joint pre-processing and assembling times were measured for each benchmark, as shown in Table 4. The correlation between Tables 3 and 4 shows that, for a given CPU, the time increases at a smaller rate than the number of assembled instructions.

For instance, in the case of MIPS, if we compare programs *negcnt* and *sort*, the number of instructions is multiplied by 5, while the time is multiplied by 2.3. This behavior is an evidence of the efficiency of the generated tools. On the other hand, for a given program tested in distinct CPUs, the time can grow at a slightly higher rate than the number of assembled instructions. For program *sort*, for instance, if we compare PIC and PowerPC, the number of instructions is multiplied by 1.3, while the time is multiplied by 2.2. This can be attributed to PowerPC's more complex instruction-set formats.

Table 3. Benchmark characterization

Benchmark	**MIPS**		**PowerPC**		**PIC**	
negcnt	154	**32**	121	**28**	150	**30**
gcd	209	**53**	198	**48**	125	**32**
int2bin	153	**36**	143	**35**	256	**36**
cast	80	**48**	87	**55**	68	**60**
fib	490	**106**	445	**96**	358	**70**
sort	2982	**154**	2744	**48**	2227	**110**

Table 4. Pre-processing and assembling times

Benchmark	**Time [s]**		
	MIPS	PowerPC	PIC
negcnt	0.030	0.035	0.020
gcd	0.030	0.050	0.020
int2bin	0.040	0.040	0.030
cast	0.040	0.050	0.030
fib	0.060	0.080	0.040
sort	0.070	0.110	0.050

5 Conclusions and Perspectives

The proposed technique and the associated tools were properly validated by means of experiments. Moreover, we have presented efficiency evidences for both the generating and the generated tools. Generation times are low enough and grow linearly with the instruction-set size. Pre-processing and assembling times grow linearly with the number of assembled instructions. Generator robustness and correctness are guaranteed by formal techniques based on context-free grammars. This allows fast and safe tool maintenance or upgrade in face of ADL changes or extensions. Generator maintainability and extensibility is enhanced by the co-validation mechanism embedded in GALS. One of our next research topics is to investigate techniques for compiler generation from an ADL description, again from a maintainability and extensibility perspective.

Acknowledgment

We would like to thank the members of the Computing Systems Laboratory at UNICAMP for their help with the ArchC package, especially Sandro Rigo and Professors Guido Araújo and Rodolfo de Azevedo.

References

1. Bergamaschi, R.: A to Z of SoCs. Brazilian Microelectronics School Tutorial (EMICRO 2002), Florianópolis, Brazil, 2002
2. Computing Systems Lab, State University of Campinas: The ArchC Architectural Description Language. Available at http://www.archc.org.
3. Open SystemC Initiative: Available at http://www.systemc.org.
4. Dalton Project: Available at http://www.cs.ucr.edu/ dalton/i8051/i8051syn/
5. Marwedel, P.: Embedded System Design. Kluwer Academic Publishers (2003)
6. Hadjiannis, G., Hanono, S., e Devadas, S.: ISDL: An instruction set description language for retargetability. In: Proc. 34th Design Automation Conference (1997) 299–302
7. Braun, G., et al.: A novel approach for flexible and consistent ADL-driven ASIP design. In: Proc. 41th Design Automation Conference (2004) 717–722
8. Mishra, P., Dutt, N.D., e Nicolau, A.: Functional abstraction driven design space exploration of heterogeneous programmable architectures. In: Proc. Int. Symposium on System Synthesis (2001) 256–261
9. Guesser, C. E.,: GALS - A Lexical and Syntactical Analyzer Generator. Research Report - Computer Science Department, Federal University of Santa Catarina, Florianópolis, SC, (2002)
10. Freerick, M.: The nML Machine Description Formalism. (1993). http://www.cs.tu-berlin.de/ mfx/dvi_docs/nml_2.dvi.gz
11. Pees, S., et al.: LISA-machine description language for cycle-accurate models of programmable DSP architectures. In: Proc. 36th Design Automation Conference, New Orleans (1999) 933–938
12. Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., Nicolau, A.: EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. Design, Automation and Test in Europe, Munich, Germany (1999) 485–490
13. Aho, A., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1988)
14. Kumari, S.: Generation of assemblers using high level processor models. Master Thesis. Department of Computer Science and Engineering - Indian Institute of Technology, Kanpur, India (2000)
15. Chiu, P.K., Fu, S.T.K.: A Generative Approach to Universal Cross Assembler Design. ACM SIGPLAN Notices (1990) 43–51
16. Wu, H., Jin, Y.: GPASM: A general purpose cross assembler for microprocessors. In: Proc. IEEE Asian Electronics Conference. (1987) 470–472
17. Tracz, W.J.: Advances in microcode support software. In: Proc. 18th Annual Workshop on Microprogramming. (1985) 57–60

Sandbridge Software Tools*

John Glossner[1,3], Sean Dorward[1], Sanjay Jinturkar[1], Mayan Moudgill[1], Erdem Hokenek[1], Michael Schulte[2], and Stamatis Vassiliadis[3]

[1] Sandbridge Technologies, 1 North Lexington Ave., White Plains, NY, 10512, USA
jglossner@sandbridgetech.com
http://www.sandbridgetech.com

[2] University of Wisconsin, Dept. of ECE, 1415 Engineering Drive, Madison, WI, 53706, USA
schulte@engr.wisc.edu
http://mesa.ece.wisc.edu

[3] Delft University of Technology, Computer Engineering Lab, Delft, The Netherlands
s.vassiliadis@its.tudelft.nl
http://ce.et.tudelft.nl

Abstract. We describe the generation of the simulation environment for the Sandbridge Sandblaster multithreaded processor. The processor model is described using the Sandblaster architecture Description Language (SaDL), which is implemented as python objects. Specific processor implementations of the simulation environment are generated by calling the python objects. Using just-in-time compiler technology, we dynamically compile an executing program and processor model to a target platform, providing fast interactive responses with accelerated simulation capability. Using this approach, we simulate up to 100 million instructions per second on a 1 GHz Pentium processor. This allows the system programmer to prototype many applications in real-time within the simulation environment, providing a dramatic increase in productivity and allowing flexible hardware-software trade-offs.

1 Introduction

The *architecture* of a computer system is the minimal set of properties that determine what programs will run and what results they will produce 0. It is the contract between the programmer and the hardware. Every computer is an interpreter of its *machine language* - that representation of programs that resides in memory and is interpreted (executed) directly by the (host) hardware. A *simulator* is an interpreter of a machine language where the representation of programs resides in memory but is not directly executed by host hardware. Historically, three types of architectural simulators have been identified. An *interpreter* consists of a program executing on a computer, where each machine language instruction is executed on a model of a target architecture running on the host computer. Because interpreted simulators tend to execute slowly, compiled simulators have been developed. A *statically compiled simulator* first translates both the

* This paper has been presented at the SAMOS IV workshop 2004.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 269–278, 2005.

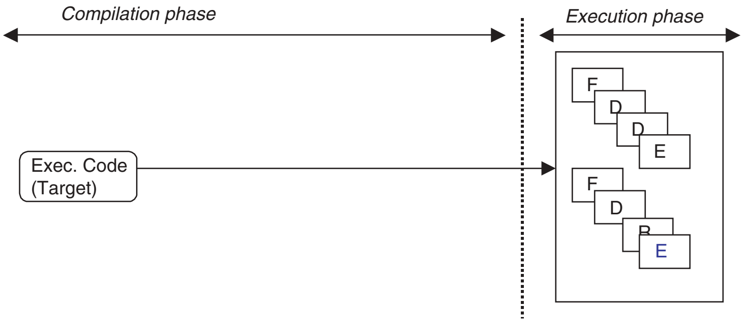


Fig. 1. Interpreted Simulation

program and the architecture model into the host computer's machine language. A *dynamically compiled* (or just-in-time) *simulator* either starts execution as an interpreter, but judiciously chooses functions that may be translated during execution into a directly executable host program, or begins by translating at the start of the host execution.

Instructions set simulators commonly used for application code development are cycle-count accurate in nature. They use an architecture description of the underlying processor and provide close to accurate cycle counts, but typically do not model external memories, peripherals, or asynchronous interrupts. However, the information provided by them is generally sufficient to develop the prototype application.

Figure 1 shows an interpreted simulation system. Executable code is generated for a target platform. During the execution phase, a software interpreter running on the host interprets (simulates) the target platform executable. The simulator models the target architecture, may mimic the implementation pipeline, and has data structures to reflect the machine resources such as registers. The simulator contains a main driver loop, which performs the *fetch*, *decode*, *data read*, *execute* and *write back* operations for each instruction in the target executable code.

An interpreted simulator has performance limitations. Actions such as instruction fetch, decode, and operand fetch are repeated for every execution of the target instruction. The instruction decode is implemented with a number of conditional statements within the main driver loop of the simulator. This adds significant overhead especially considering all combinations of opcodes and operands must be distinguished. In addition, the execution of the target instruction requires the update of several data structures that mimic the target resources, such as registers, in the simulator.

Figure 2 shows a statically compiled simulation system. In this technique, the simulator takes advantage of the any *a priori* knowledge of the target executable and performs some of the activities at compile time instead of execution time. Using this approach, a simulation compiler generates host code for instruction fetch, decode and operand reads at compile time. As an end product, it generates an application specific host binary in which only the execute phase of the target processor is unresolved at compile time. This binary is expected to execute faster, as repetitive actions have been taken care of at compile time.

While this approach addresses some of the issues with interpretive simulators, there are further limitations. First, the simulation compilers typically generate C code, which

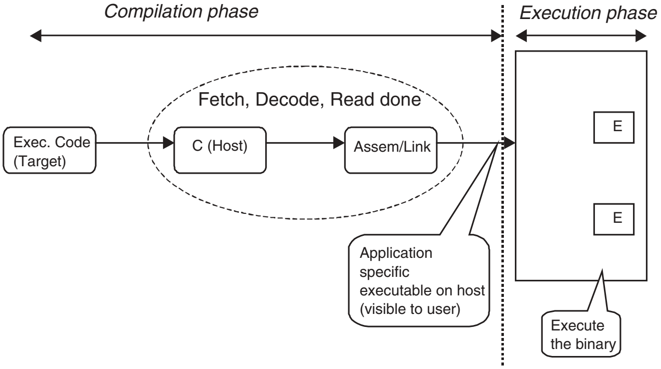


Fig. 2. Statically Compiled Simulation

is then converted to object code using the standard *compile→assemble→link* path. Depending on the size of the generated C code, the file I/O needed to scan and parse the program could well reduce the benefits gained by taking the compiled simulation approach. The approach is also limited by the idiosyncrasies of the host compiler such as the number of labels allowed in a source file, size of switch statements etc. Some of these could be addressed by directly generating object code - however, the overhead of writing the application specific executable file to the disc and then re-reading it during the execution phase still exists. In addition, depending on the underlying host, the application-specific executable (which is visible to the user) may not be portable to another host due to different libraries, instruction sets, etc.

Figure 3 shows the dynamically compiled simulation approach. In this approach, target instructions are translated into equivalent host instructions (executable code) at the beginning of execution time. The host instructions are then executed at the end of the translation phase. This approach eliminates the overhead of repetitive target instruction fetch, decode and operand read in the interpretive simulation model. By directly generating host executable code, it eliminates the overhead of the compile, assemble, and link path and the associated file I/O that is present in the compiled simulation ap-

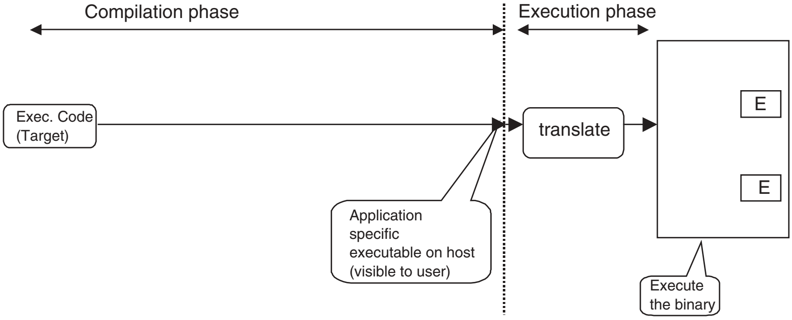


Fig. 3. Dynamically Compiled Simulation

proach. This approach also ensures that the target executable file remains portable, as it is the only executable file visible to the user and the responsibility of converting it to host binary has been transferred to the simulator.

This paper is organized as follows. In Section 2, we present a transparent multithreaded architecture that provides for scalable implementations. In Section 3, we describe how our toolchain is generated. In Section 4, we provide simulation results and discuss related work. In Section 5, we draw conclusions.

2 Sandblaster Processor

An architectural function is *transparent* if its implementation does not produce any architecturally visible side effects. Generally, it is desirable to have transparent implementations. Many architectures, however, are not transparent. When modeling an architecture, it is often desirable to model a specific implementation's performance. Because generating tools for a multiplicity of architectures and implementations is resource intensive, architecture description languages have been developed [2][3]. A characteristic of this approach has been the generation of both the tool chain and hardware description.

Sandbridge Technologies has developed the Sandblaster architecture for convergence devices [4][5]. Just as handsets are converging to multimedia multiprotocol systems, the Sandblaster architecture supports the datatypes necessary for convergence devices including RISC control, DSP, and Java code. As shown in Fig. 4, the Sandblaster multithreaded processor design includes a unique combination of modern techniques

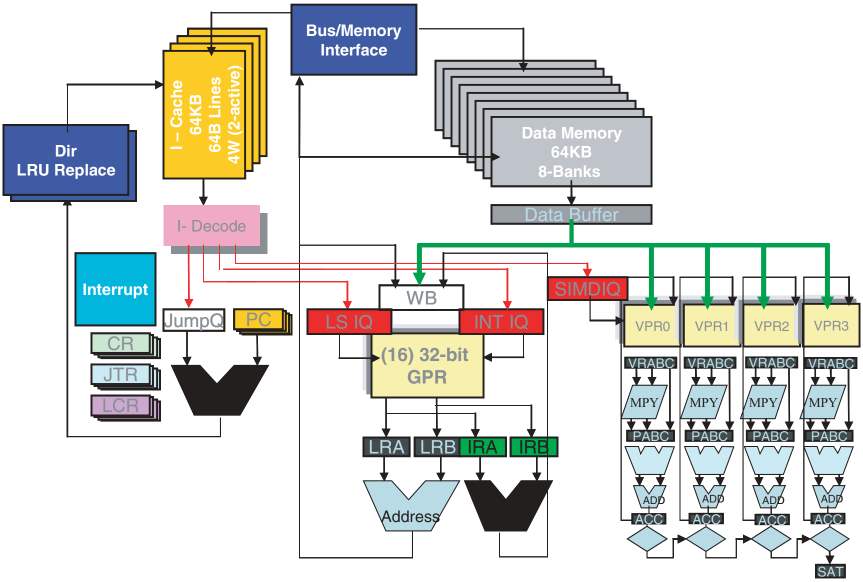


Fig. 4. Sandblaster Multithreaded Processor

such as a SIMD Vector unit, a parallel reduction unit, and a RISC-based integer unit. Each processor core provides support for concurrent execution of up to eight threads.

A RISC-based integer execution unit, depicted in the center of Fig. 4, assists with control processing. Physical layer processing often consists of control structures with compute-intensive inner loops. A baseband processor must deal with both integer and fractional datatypes. For the control code, a register file with 16 32-bit entries per thread provides for very efficient control processing. Common integer datatypes are typically stored in the register file. This allows for branch bounds to be computed and addresses to be generated efficiently.

Intensive DSP physical layer processing is performed in the SIMD Vector unit depicted on the right side of Fig. 4. Each cycle, four 16-bit vector elements may be loaded into the Vector File, while four pairs of 16-bit vector elements are multiplied and then reduced (e.g. summed), with saturation after each operation. The branch bound may also be computed and the instruction repeated until the entire vector is processed. Thus, retiring four elements of a saturating dot product and looping may be specified in as little as 64-bits, which compares very favorably to VLIW implementations.

An important power consideration is that the Vector File contains a single write port. This is in distinct contrast to VLIW implementations that must specify an independent write port for each operation in the VLIW instruction. Consequently, VLIW instructions, which are often up to 256-bits, may require register files with eight or more write ports. Since write ports contribute significantly to power dissipation, minimizing them is an important consideration in handset design.

To enable physical layer processing in software, the processor supports many levels of parallelism. Thread-level parallelism is supported by providing hardware for up to eight independent programs to be simultaneously active on a single Sandblaster core. This minimizes the latency in physical layer processing. Since many algorithms have stringent requirements on response time, multithreading is an integral technique in reducing latencies.

In addition to thread-level parallelism, the processor also supports data-level parallelism through the use of the SIMD Vector unit. In the inner kernel of signal processing or baseband routines, the computations appear as vector operations of moderate length. Filters, FFTs, convolutions, etc., all can be specified in this manner. Efficient, low-power support for data-level parallelism effectively accelerates inner loop signal processing.

To accelerate control code, the processor supports issuing multiple operations per cycle. Since control code often limits overall program speedup (e.g. Amdahl's Law), it is helpful to allow control code and vector code to be overlapped. This is provided through a compound instruction set. The Sandblaster core provides instruction level parallelism by allowing multiple operations to issue in parallel. Thus, a branch, an integer, and a vector operation may all issue simultaneously. In addition, many compound operations are specified within an instruction class such as load with update, and branch with compare.

Finally, the SB3000 product includes four Sandblaster processor cores per chip to provide enough computational capability to execute complete WCDMA baseband processing in software in real-time.

Future 3G wireless systems will make significant use of Java. A number of carriers are already providing Java-based services and may require all 3G systems to support Java [6]. Java, which is similar to C++, is designed for general-purpose object-oriented programming [7]. An appeal for the usage of such a language is its "write once, run anywhere" philosophy [7]. This is accomplished by providing a Java Virtual Machine (JVM) interpreter and runtime support for each platform [8][9].

JVM translation designers have used both software and hardware methods to execute Java bytecode. The advantage of software execution is flexibility. The advantage of hardware execution is performance. The Delft-Java architecture, designed in 1996, introduced the concept of dynamic translation of Java code into a multithreaded RISC-based machine with Vector SIMD DSP operations [10][11]. The important property of Java bytecode that facilitated this translation is the statically determinable type state [7]. The Sandbridge approach is a unique combination of both hardware and software support for Java execution.

A challenge of visible pipeline machines (e.g. most DSPs and VLIW processors) is interrupt response latency. Visible memory pipeline effects in highly parallel inner loops (e.g. a load instruction followed by another load instruction) are not typically interruptible because the processor state cannot be restored. This requires programmers to break apart loops so that worst case timings and maximum system latencies are acceptable. This convolutes the source code and may even require source code changes between processor generations.

The Sandblaster core allows any instruction from any thread to be interrupted on any processor cycle. This is critical to real-time constraints imposed by physical layer processing. The processor also provides special hardware support for a specific thread unit to interrupt another thread unit with very low latency. This low-latency cross-thread interrupt capability enables fast response to time critical events.

3 Tool Chain Generation

Figure 5 shows the Sandblaster tool chain generation. The platform is programmed in a high-level language such as C, C++, or Java. The program is then translated using an internally developed supercomputer class vectorizing, parallelizing compiler. The tools are driven by a parameterized resource model of the architecture that may be programmatically generated for a variety of implementations and organizations. The source input to the tools, called the Sandbridge architecture Description Language (SaDL), is a collection of python source files that guide the generation and optimization of the input program and simulator. The compiler is retargetable in the sense that it is able to handle multiple possible implementations specified in SaDL and produce an object file for each implementation. The platform also supports many standard libraries (e.g. libc, math, etc.) that may be referenced by the C program. The compiler generates an object file optimized for the Sandblaster architecture.

The tools are then capable of producing dynamic and static simulators. A binary translator/compiler is invoked on the host simulation platform. The inputs to the translator are the object file produced by the Sandblaster compiler and the SaDL description of the processor. From these inputs, it is possible to produce a statically compiled simulation file. If the host computer is an x86 platform, the translator may directly produce x86 optimized code. If the host computer is a non-x86 platform, the binary translator produces a C file that may subsequently be processed using a native compiler (e.g. gcc).

For the dynamically compiled simulator, the object file is translated into x86 assembly code during the start of the simulation. In single-threaded execution, the entire program is translated and executed, removing the requirement for fetch-decode-read operations for all instructions.

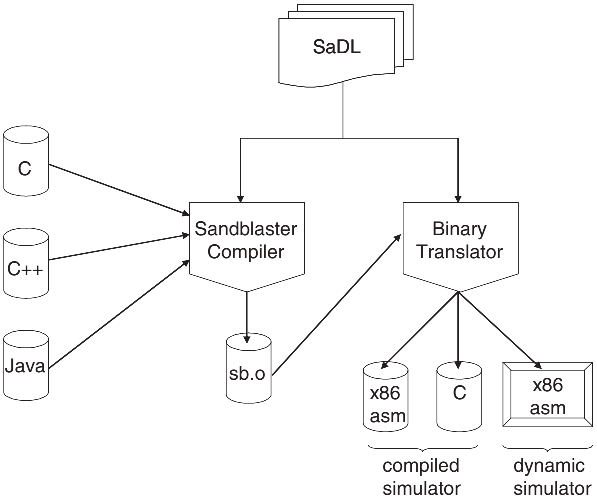


Fig. 5. Tool Chain Generation

Dynamically compiled single threaded simulation translation is done at the beginning of the execution phase. Regions of target executable code are created. For each compound instruction in the region, equivalent host executable code is generated. Within each instruction, sophisticated analysis and optimizations are performed to reorder the host instructions to satisfy constraints. When changes of control are present, the code is modified to the proper address. The resulting translated code is then executed.

The SaDL description is based on the philosophy of abstracting out the Sandblaster architecture and implementation specific details into a single database. The information stored in the architectural description file can be used by various parts of the tool chain. The goal is to keep a single copy of the information and replicate it automatically as and when needed. The key part of the architectural description language is a set of python files which abstract the common information. These files keep information about each opcode on the Sandblaster processor. The description of an opcode contains a number of attributes - the opcode name, opcode number, format, and the registers. In addition, it contains the appropriate host code to be generated for the particular opcode. These description files are then processed by a generator to automatically produce the C code and documentation. The produced C code is used by our just-in-time simulator and other tools.

```
shr = opcode("shr", opcode = 0xa4, format = (Rt, Ra, Rb),
      resources = binop_resources(),
      jx86_exec = jx86_shu_body("shrl") + jx86_intop_wback(),
      doc_full = "Shift Right",
      doc_stmt = [
            EOp( EGP("rt"), "<-", EOp(EGP("ra"), ">>", EGP("rb")) )],
      doc_long = "The target integer register, rt, is set to the
      value of the input register ra right shifted by the value of rb;
      0s are shifted in to the high bits.")
```

Fig. 6. Example SaDL Python Descriptions

Figure 6 shows an example of an opcode entry in our architecture description language. It contains the opcode name, number, format and the input resources. It also has calls to the functions (jx86_exec statement) that are called to implement the operation on the host platform. It contains both the mathematical description (doc_stmt) and the English description (doc_long) to document the opcode

Dynamically compiled multithreaded simulation is more complex than the single-thread case because multiple program counters must also be accounted for. Rather than translating the entire object file as one monolithic block of code with embedded instruction transitions, and then executing it, in the multithreaded case we begin by translating each compound instruction on an instruction-by-instruction basis. A separate piece of code manages the multiple pc's, selects the thread to execute, and performs calls to the translated instruction(s) to perform the actual operation. The fetch cycle for each thread must be taken into account based on the scheduling policy defined in the SaDL implementation parameters. Properly speaking, the thread scheduling policy need not be considered for logical correctness; however, it facilitates program debugging. Although fetching with multiple program counters has an effect on simulation performance, compiled dynamic multithreaded simulation is still significantly faster than interpreted simulation.

When the simulator encounters a particular opcode during simulation, it calls the appropriate C function (generated from the processed architectural description files) for that opcode, makes a syntactic and semantic check and generates the host code to be executed on the host processor. By using this approach, all the modifications to the architecture description are limited to a small set of files. This minimizes errors and maximizes productivity.

4 Results and Related Work

Figure 7 shows the simulation performance of the ETSI AMR speech encoder on out-of-the-box C code [13]. The simulation speed was measured by taking the wall clock time needed to simulate a certain number of cycles on a 1 GHz Pentium. The results show that the simulation performance of a single threaded fully optimized and vectorized AMR encoder is 25 million instructions per second. The performance degrades to 15 MIPS for 8 threads and then very slightly for additional threads. The degradation is due to the overhead of simulating multiple instruction counters.

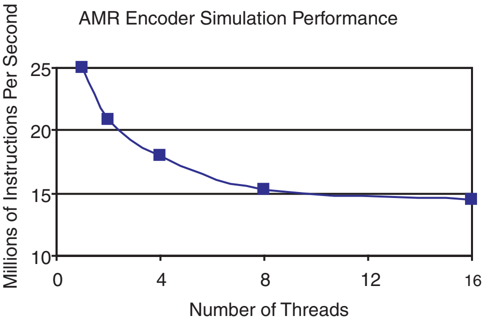


Fig. 7. Simulation Results

Previously, we have compared the simulation speed of our approach with that of other DSP simulators [14]. Our approach is up to four orders of magnitude faster than current DSP simulators. Comparatively, we can simulate in real-time on a simulation model of the processor while other approaches have difficulty achieving real-time performance on their own native platform. This provides a tremendous advantage in prototyping algorithms.

Automatic DSP simulation generation from a C++-based class library was discussed in [15]. Automatic generation of both compiled and interpretive simulators was discussed in [16]. Compiled simulation for programmable DSP architectures to increase simulation performance was introduced in [17]. This was extended to cycle accurate models of pipelined processors in [3]. A general purpose MIPS simulator was discussed in [18]. The ability to dynamically translate snippets of target code to host code at execution time was used in Shade [19]. However, unlike Shade, our approach generates code for the entire application and is targeted towards compound instruction set architectures.

5 Conclusions

We have presented a methodology for generating and simulating the Sandblaster architecture using the SaDL architecture description language. On a 1 GHz Pentium processor, our dynamically compiled simulator is capable of simulating up to 100 million instructions per second for lightly optimized code. Fully optimized production DSP code simulates at roughly 25 million instructions per second and fully optimized multithreaded simulation provides nearly 15 million instructions per second.

References

1. Blaauw, G., Brooks, F.: Computer Architecture: Concepts and Evolution. Addison-Wesley, Reading, Massachusetts (1997)
2. Fauth, A., Knoll, A.: Automated generation of DSP program development tools using a machine description formalism. In: IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-93). Volume 1. (1993) 457–460

3. Pees, S., Hoffmann, A., Zivojnovic, V., Meyr, H.: Lisa - machine description language for cycle accurate models of programmable DSP architectures. In: Proceedings of the 36th Design Automation Conference. (1999) 933–938
4. Glossner, J., Raja, T., Hokenek, E., Moudgill, M.: A multithreaded processor architecture for SDR. The Proceedings of the Korean Institute of Communication Sciences **19** (2002) 70–84
5. Glossner, J., Hokenek, E., Moudgill, M.: Multithreaded processor for software defined radio. In: Proceedings of the 2002 Software Defined Radio Technical Conference. Volume I, San Diego, CA (2002) 195–199
6. Yoshida, J.: Java chip vendors set for cellular skirmish. EE Times (2001)
7. Gosling, J.: Java intermediate bytecodes. In: ACM SIGNPLAN Workshop on Intermediate Representation (IR95). (1995) 111–118
8. Gosling, J., McGilton, H.: The Java language environment: A white paper. Sun Microsystems Press (1995)
9. Lindholm, T., Yellin, F.: Inside the Java virtual machine. Unix Review **15** (1997) 31–39
10. Glossner, J., Vassiliadis, S.: The Delft-Java engine: An introduction. In: Third International Euro-Par Conference (Euro-Par '97), Passau, Germany (1997) 776–770
11. Glossner, J., S. Vassiliadis, S.: Delft-Java dynamic translation. In: Proceedings of the 25th EUROMICRO Conference (EUROMICRO '99), Milan, Italy (1999)
12. Ebcioglu, K., Altman, E., Hokenek, E.: A Java ILP machine based on fast dynamic compilation. In: IEEE MASCOTS International Workshop on Security and Efficiency Aspects of Java, Eilat, Israel (1997)
13. European Telecommunications Standards Institute: Digital cellular telecommunications system, ANSI-C code for adaptive multi rate speech traffic channel (AMR). (Ver 7.1.0) (1999)
14. Glossner, J., Iancu, D., Lu, J., Hokenek, E., Moudgill, M.: A software defined communications baseband design. IEEE Communications Magazine **41** (2003) 120–128
15. Parson, D. Beatty, P., Glossner, J., Schlieder, B.: A framework for simulating heterogeneous virtual processors. In: Proceedings of the 32nd Annual Simulation Conference San Diego, CA (1999) 58-67
16. Leupers, R., Elste, J., Landwehr, B.: Generation of interpretive and compiled instruction set simulators, In: Proceedings of the ASP-DAC '99, Wanchai, Hong Kong **1** (1999) 339–342
17. Zivojnovic, V., Tjiamg, S., Meyr, H.: Compiled simulation of programmable DSP architectures. In: Proceedings of the IEEE Workshop on VLSI Signal Processing, Sakai, Osaka (1995) 187–196
18. Zhu, J., Gajski, D.: An ultra-fast instruction set simulator. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **10** (2002) 363–373
19. Cmelik, R.: Shade: A fast instruction-set simulator for execution profiling. Technical Report UWCSE 93-06-06, Univ. Of Washington

A Hardware Accelerator for Controlling Access to Multiple-Unit Resources in Safety/Time-Critical Systems

Philippe Marchand[1] and Purnendu Sinha[2]

[1] Dept. of ECE Concordia University, Montreal, Canada
p_marcha@ece.concordia.ca
[2] Indian Institute of Information Technology Bangalore, India
psinha@iiitb.ac.in

Abstract. In multitasking, priority-driven systems, resource access-control protocols such as Priority Ceiling Protocol (PCP) reduce the undesirable effects of resource contention. In general, software implementation of these protocols entails costly computations that can degrade the system performance to unacceptable levels. In this paper, we present the design for a hardware-accelerator to execute the PCP functionality for controlling access to multiple-unit resources and illustrate that the proposed implementation accelerates the execution time by a factor of up to 30.

1 Introduction

In a multitasking uniprocessor environment, improper resource sharing among tasks could lead to significant performance penalties as well as severe adverse effects. Priority Ceiling Protocol (PCP) [1] [9] is a resource management protocol that prohibits the occurrence of deadlocks and minimizes priority inversion in such an environment. Deadlocks and priority inversion are serious problems that can have catastrophic effects in safety-critical real-time systems. Our experience has been that software implementations of these resource management policies account for a significant portion of performance degradation in such systems. In order to alleviate system's degraded performance we have designed and implemented a hard accelerator to execute the functionalities of the PCP handling *multiple-unit* resources. Both software and hardware implementations of the protocol have been integrated with the μC/OS-II [7] operating system running on the AVR ATmega103L [2] microcontroller implemented on a Xilinx Virtex XCV300 [12] FPGA board [2].

We first provide a brief description of previous work accomplished in the field of accelerators and give a theoretical background explaining PCP for multiple-unit resource access. We then discuss the methodology adopted as well as the software and hardware implementations developed. Finally, we present experimental results comparing the performance of both designs.

[1] There are two variants of PCP [9]: Original Ceiling Priority Protocol (OCPP) and Immediate Ceiling Priority Protocol (ICPP). We utilize OCPP for multiple-unit resources and refer to it as PCP in the paper.

[2] The rationale for using μC/OS-II, ATMega103L and XCV300 is due to their easy accessibility.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 279–288, 2005.

2 Related Work

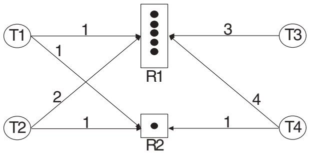
Significant work has been done in the area of software/hardware co-design and in the more specific area of hardware accelerators for embedded computing. Fundamental software/hardware co-design issues, namely partitioning and scheduling, have been outlined [5] and examples of accelerator design has been covered in [6, 11]. Several authors have proposed variations to the deadlock detection and avoidance algorithms to improve their execution time when executed in software [3, 4]. Mooney et al. developed a hardware accelerator for deadlock detection based on resource allocation graphs in multiprocessor systems in [10]. Resource management acceleration using hardware has also been explored by using a system-on-a-chip lock cache to execute the Immediate Ceiling Priority Protocol (ICPP) [9] for *single-unit* of resources on a *multiprocessor system* running the Atalanta-RTOS, achieving impressive performance gains [1]. In this project, we developed the original PCP [9] (as opposed to ICPP) for resources of *multiple-units* in a multitasking, *uniprocessor* environment.

3 Theoretical Background

A deadlock can occur when a task is waiting on a resource that it can never acquire. This situation often arises when more than one task in a system must acquire more than one resource at one specific time. For example, one of the simplest deadlock situations occurs when a task $T1$ owns resource $R1$ and needs resource $R2$ to progress and conversely, at the same moment task $T2$ owns $R2$ and needs $R1$ to progress. In order to prevent deadlocks, a resource management scheme such as PCP must keep a record of the state of each resource. Using this information, it can determine if the allocation of a particular resource to a given task would cause a deadlock situation.

Priority inversion occurs when a lower priority task blocks a higher priority task, and can be is triggered by the sequencing of the resource allocations. Consider the trivial condition where a low priority task $T1$ acquires a resource that is also used by a high priority task, $T2$. If $T2$ blocks because it cannot acquire this resource, priority inversion occurs when $T1$ runs. The problem becomes more acute if additional tasks with intermediate priorities are executing in the system since these would preempt $T1$ and in the process further delay the execution of the higher priority task $T2$.

Next, we briefly describe the working principles of the PCP for controlling access to multiple-unit resources. For details, we refer the reader to [8, 9]. PCP implements deadlock avoidance by assigning a priority ceiling (PC) to every resource in the system. The PC of resource R is defined as the priority of the highest priority task that uses R. In an environment where there are multiple instances of the same resource, the PC becomes a function of not only the resource type but also of the remaining number of instances of that resource. Given a resource R that has N units, the PC when there are $n \leq N$ free units of R is equal to the priority of the highest priority task that uses at least n instances of R. For example, given the following resource allocation graph of Fig. 1, where task $T1$ to $T4$ are indexed in decreasing order of their priorities, we obtain the Priority Ceiling Array of Fig. 1 that displays the PC as a function of resource and units left. For instance, if there are 2 units of $R1$ left, the PC will be $\Pi(R1,2) = 3$.



Resource	Number of units	$\Pi(R,k)$					
		Number of units left, k					
		0	1	2	3	4	5
R1	5	1	2	3	4	-	-
R2	1	1	-	-	-	-	-

Fig. 1. Resource allocation graph and corresponding Priority Ceiling Array

PCP states that a task can only acquire a resource if its priority is greater than the PC of every resource instance currently held by other tasks. PCP protects against priority inversion by executing Priority Inheritance, which seeks to correlate the time a task is kept waiting on a resource to the relative importance given to it by its assigned priority. If a task $T1$ is blocked waiting on units of a resource, the task owning those units, $T2$ for instance, will acquire $T1$'s priority if the priority of $T2$ is less than the priority of $T1$. To implement PCP, we define the system priority ceiling, *SysPC*, which is the highest priority ceiling of the currently obtained resource units:

$$SysPC = \max\{\forall \mathrm{PC}\,(\Pi(R_{ACQUIRED}, n))\}$$

Also, the system task, *SysTask*, is defined as the task that owns the resource with a Priority Ceiling equal to *SysPC*.

The rule used to determine whether a task $T1$ with priority π_1 can obtain resource units is:

$$(\pi_1 > SysPC) \text{ or } (T1 = SysTask) \tag{1}$$

This scheme allows the PCP algorithm to be implemented in either hardware or software. A list is kept of the resource units presently acquired and their corresponding owning task. From this list, the *SysPC* and *SysTask* variables can be easily computed and used to evaluate (1) when a task seeks to obtain resource instances.

Priority Inheritance will be carried out when a task fails to acquire a resource and effectively becomes blocked by the *SysTask*. In this case, if the priority of the blocked task is higher than the priority of the *SysTask*, the tasks will exchange priorities.

When a task fails to obtain requested resource units either because it does not meet (1), or there simply aren't enough free instances, a task is blocked waiting on the resource and must go in the wait list. The scheduler can then move a task from the wait list to the ready list when the blocking resource instances become available. In order to implement PCP, tasks and resources, in this case semaphores, are uniquely identified. These identification fields are entered into nodes of the list when a task acquires semaphore units, and used to remove or modify a node when it releases units. Each node also contains the priority ceiling and amount of semaphore instances.

A hardware implementation of ICPP has been introduced in [1]. Note that ICPP takes a more straightforward approach and raises the priority of a process to the priority ceiling of the resource it has just locked. ICPP is easier to implement than OCPP as blocking relationships need not be monitored. Although ICPP is simpler to implement and reduces the amount of context switches, this protocol can increase the occurrence of priority inversion. By immediately raising the priority of a task to the priority ceiling of the resource it has just acquired, higher priority tasks that do not utilize resources could be needlessly blocked by a task that has just acquired resource units with a high PC. This further highlights our approach as compared to [1].

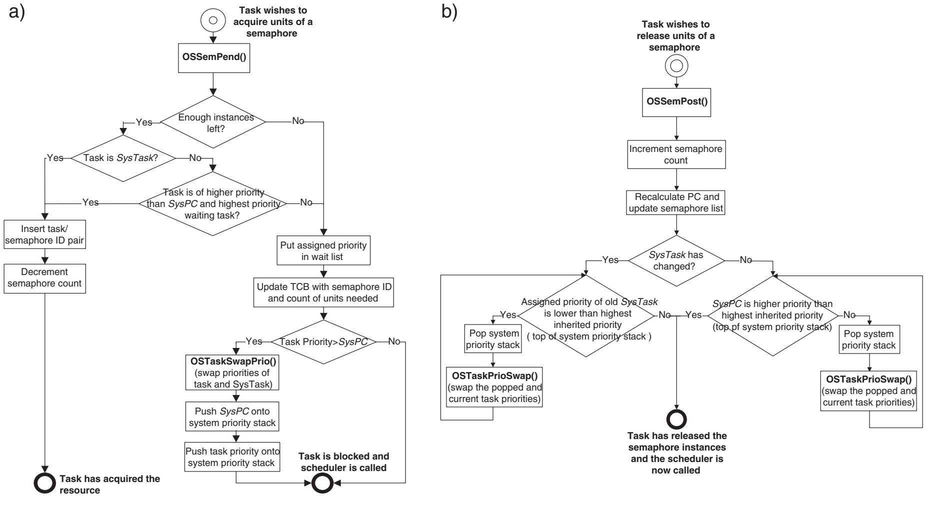
4 Methodology

In our implementation, PCP has been decomposed into functional blocks: the semaphore *acquire* and *release* functions, as well as new task management features to support PCP, which have all been implemented in both hardware and software. The target platform, running the μC/OS-II operating system, supports up to 64 tasks of unique, 8-bit priorities that are sequenced in reverse numerical order [7]. To facilitate both the hardware and software implementations, a task is given an ID that is also its assigned priority. In the rest of the text, a task ID is synonymous with the task's assigned priority.

4.1 Outline of Acquire Function

Figure 2a shows the flowchart which represents the sequence of actions that take place when a task wishes to acquire semaphore instances, starting with the function call *OSSemPend*(), which encapsulates the whole operation. If there are enough semaphore instances left *and* the task is the *SysTask* or it has a higher priority than *SysPC* and the highest priority blocked task, the task acquires the semaphore units and can progress.

On the other hand, if it fails this condition, the task becomes blocked waiting on that particular resource. At this point, priority inheritance will be executed if the task has a higher priority than the priority of the *SysTask* by exchanging priorities with it by calling *OSTaskSwapPrio*(). At the same time, these swapped priorities are pushed onto the system priority stack, which holds the priorities of every task that have exchanged priorities. Finally, since the task is now blocked, its assigned priority is removed from



a)
Task wishes to acquire units of a semaphore
OSSemPend()
Enough instances left?
Yes
No
Task is SysTask?
Yes
No
Task is of higher priority than SysPC and highest priority waiting task?
Yes
No
Insert task/ semaphore ID pair
Decrement semaphore count
Put assigned priority in wait list
Update TCB with semaphore ID and count of units needed
Task Priority>SysPC
Yes
No
OSTaskSwapPrio() (swap priorities of task and SysTask)
Push SysPC onto system priority stack
Push task priority onto system priority stack
Task is blocked and scheduler is called
Task has acquired the resource
b)
Task wishes to release units of a semaphore
OSSemPost()
Increment semaphore count
Recalculate PC and update semaphore list
SysTask has changed?
Yes
No
Assigned priority of old SysTask is lower than highest inherited priority (top of system priority stack)
Yes
No
SysPC is higher priority than highest inherited priority (top pf system priority stack)
Yes
No
Pop system priority stack
OSTaskPrioSwap() (swap the popped and current task priorities)
Pop system priority stack
OSTaskPrioSwap() (swap the popped and current task priorities)
Task has released the semaphore instances and the scheduler is now called

Fig. 2. Flowchart representing steps taken when (a) acquiring and (b) releasing a semaphore

the ready list and inserted into the wait list. The Task Control Block (TCB) of the blocked task is updated with the ID and count of the semaphore it is waiting on.

4.2 Outline of Release Function

When releasing semaphore instances, the basic steps include modifying the semaphore list and reversing priority inheritance. Figure 2b shows the flowchart representing the sequence of actions that occur during this operation. Function *OSSemPost*() is called, the semaphore count is incremented, and the task/semaphore node is updated with a new count and PC field, or removed completely if the task owns no more instances of the semaphore. The ”reversing” of Priority Inheritance is accomplished by exploiting the fact that only a task that is currently the *SysTask* can inherit a higher priority and yield an inherited priority. Therefore, if a task is no longer the *SysTask* after releasing semaphore instances (in other words, the *SysTask* has changed), it must ”give up” any inherited priorities. In this situation, we know a task has inherited a priority if its assigned priority is lower than the priority on top of the system priority stack. Therefore, the stack is popped and function *OSTaskPrioSwap*() is called to swap the priorities of the task with the assigned priority equal to the popped priority and the current task. This is seen on the left branch of the chart of Figure 2b. The stack popping and priority swapping is repeated for any other priorities inherited by the current task. The other situation occurs if a task remains the *SysTask* after releasing semaphore instances, but at a lower *SysPC* value. In this case, the task gives up any inherited priorities that are of higher value than the *SysPC* value.

4.3 Outline of Scheduler Execution

The task scheduler is modified to support PCP because it must now work with a ready list and the newly added wait list. The highest priority tasks of both the ready and wait list are obtained and compared. If the latter’s priority is higher and it can obtain the semaphore instances it was blocked on, this task now becomes the running task. The task is therefore taken out of the wait list, put into ready list and executed by context switching to it. If the highest priority ready task is of higher priority, or if the highest priority wait task cannot obtain the blocking semaphore instances, the task from the ready list will run.

5 Software Implementation

The most important implementation decision of the software PCP is the choice of the data structure to hold the semaphore/owning task list. There are two criteria to consider: the cost of inserting and removing an entry and the price of determining the entry with the highest PC that holds the *SysTask* and *SysPC* values. We chose an ordered circular linked list where the first entry will have the highest PC. It is then easy to evaluate (1) each time a task wishes to acquire resource units. The disadvantage of this implementation is the time it takes to search the list when adding or removing entries, a cost that is proportional to the number of entries and executes in the order of $O(n)$, where n is the number of task/resource pairs that currently exist in the system. Utilizing an array or hash type data structure would have alleviated this performance drawback at the cost of having to reanalyze the data structure to find the new *SysTask* entry every time the entry with the highest PC was removed. For example, one option would have been to uniquely identify each entry with a combination of semaphore ID and owning task ID, in effect giving us a two-dimensional matrix, of size $M \times N$, where M is the number of tasks and N the number of semaphores in the system, which executes in the order of $\mathrm{O}(N \times M)$ to find the entry with the highest PC. For a system with many tasks and semaphores, this search would become quite expensive and would far outweigh the cost of the linked list implementation.

The performance drawbacks of the software implementation arise from the fact that priority ceiling values are not unique: several task/semaphore pairs can have the same PC value, making it costly to determine the pair or pairs with the highest PC. Task scheduling is an equivalent problem that is avoided by an RTOS that supports tasks of unique priorities: the scheduler is able to easily store the list of ready tasks and quickly determine the highest priority task ready to run. For example, the μC/OS-II scheduler stores the ready list in an 8-bit variable and uses a bit-map technique to determine the highest priority of the ready list with just two non-looping, high-level language instructions [7]. Introducing the software implementation of PCP into a RTOS that ensures low overhead by foregoing the use of costly data structures might degrade performance to unacceptable levels. The proposed solution is to implement PCP in hardware and use parallelism to avoid the performance drawbacks of the software implementation.

6 Hardware Implementation

The software implementation of the PCP has been ported to hardware by developing an accelerator that is a separate entity from the CPU. Since the AVR ATmega103L microcontroller uses port-based interfacing, the accelerator is addressed as a peripheral and communicates with the CPU via I/O Ports. At the heart of the accelerator is a register file used to hold the list of semaphore/owning task ID pairs that are stored in a linked-list with the software implementation. The system works as follows: when a task acquires or releases semaphore units, the accelerator updates the register file. If a task calls the *Acquire* function and is not granted the semaphore instances, its ID is put into the wait list that is stored in the accelerator. When a task *Releases* semaphore instances, the accelerator direct the priority inheritance procedure, telling the OS which priorities to swap. The accelerator will also determine if the highest priority waiting task can acquire the blocking semaphore instances when the scheduler is called. Finally, the accelerator must locally store the semaphore counts and the Priority Ceiling Array.

6.1 Accelerator Datapath

As previously stated, the main part of the datapath is the register file module that holds the semaphore/task ID pairs, comparing them in parallel fashion. Figure 3 shows this section of the datapath. The register file module consists of 32 entries, each containing a semaphore ID, task ID, semaphore count and PC field. On the right of Fig. 3, 8-bit comparators and multiplexers connected in a tree fashion are used to output signals *SysPC* and *SysTask*. The left of Fig. 3 shows 32 13-bit comparators that compare the

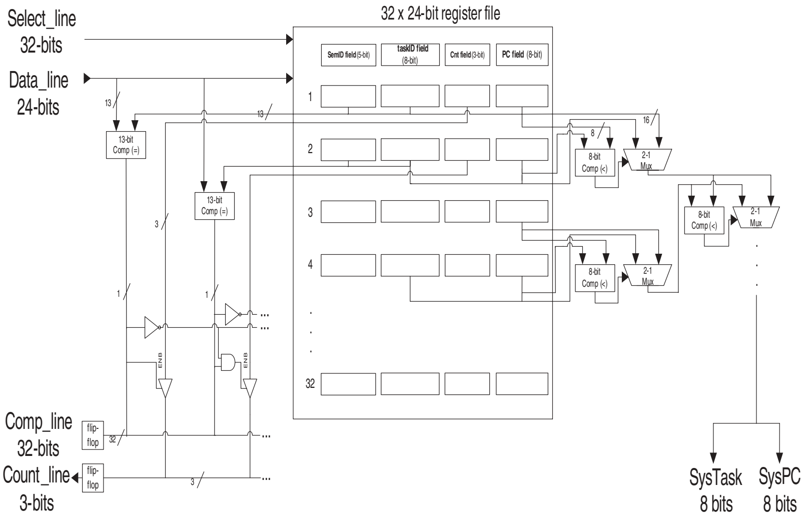


Fig. 3. Register file module with parallel logic

semaphore ID and task ID of every entry to the *Data_line*. The resulting 32 bits, identifying which entries have the same value as the *Data_line*, are stored in flip-flops and outputted by signal *Comp_line*. The outputs of the comparators are also used as the enable input to 32 3-bit tri-state buffers, which make up the *Count_line*. Thus, this signal will yield the count of the entry whose task and semaphore ID match the *Data_line*.

Figure 4 shows the top level view of the accelerator datapath. On the left are the three registers which hold the data from the CPU. On the bottom left of the diagram are the *Wait List*, *Sem_Array_Regfile* and *Cnt_Array_Regfile*. With its N 1-bit registers, the *Wait List* records which tasks are blocked (where $N \leq 64$ represents the number of tasks). The encoder translates the N bits of the registers to an 8-bit signal that holds the assigned priority of the highest priority waiting task. The *Sem_Array_Regfile* and *Cnt_Array_Regfile* contain the blocking semaphore ID and number of units needed, respectively, by the task whose ID indexes them. Two other register files, this time indexed by semaphore ID, are the *Current_Cnt_Regfile* and *Max_Cnt_Regfile*, which hold the number of units currently used and the maximum units of the selected semaphore, respectively. These numbers, as well as the count of units needed or released, are fed through two stages of adders and subtractors in order to compute the amount of semaphore units remaining. This number, along with the semaphore ID, indexes the *PC_Array_Regfile* to give the PC of the semaphore/task pair which can then be fed to the *Data_line* of the Regfile module.

The *SysTask* and *SysPC* output of the Regfile Module is supplied to the combinational logic block that is responsible for determining if semaphore units can be granted when the accelerator is executing the *Acquire function*. This block also takes in the assigned priority of the highest priority waiting task computed by the encoder of the *Wait List* and the task ID input from the CPU. With this information, the logic sets its output bit if (1) evaluates to true. The final component is the priority inheritance module responsible for implementing the system priority stack.

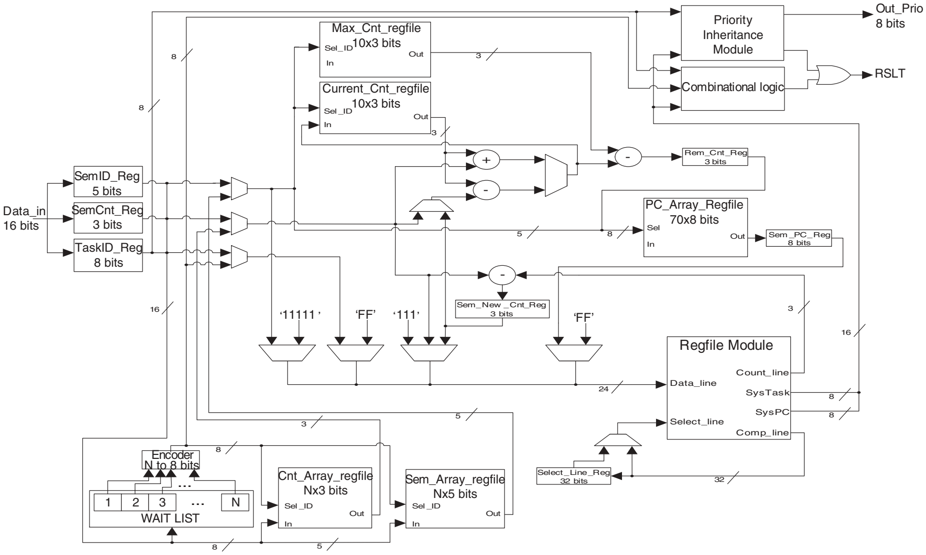


Fig. 4. Top level view of accelerator datapath

6.2 Hardware Implementation Metrics

The accelerator design was implemented in hardware on the Virtex XCV 300 FPGA by the Xilinx Design Manager software, giving us the hardware implementation metrics. The equivalent gate count is 34 871 and a maximum frequency of over 9 MHz for a design that has 32 Regfile Module entries. The AVR ATmega103L microcontroller has a maximum frequency of 6 MHz, proving that both components are compatible.

7 Results

The acceleration gains offered by the hardware implementation of the Priority Ceiling Protocol were quantified by running the PCP functionality in simulation on the soft CPU executing the AVR ATmega103L instructions. The functions performed were the Acquire and Release of semaphore instances and scheduler execution. The system consisted of 5 tasks that share 6 semaphores with a maximum of 6 units. Figure 5 shows the task execution.

Acquire Function Results. Table 1 shows the results obtained when running the Acquire semaphore function under different scenarios. The best case execution time for the software implementation occurs when the task/semaphore pair list is empty seen as point 1 on Fig. 5. The worst case happens when it is full with 29 task/semaphore pairs that must be traversed, at point 2 on Fig. 5. The accelerator, on the other hand, executes the *Acquire* function in constant time. We have also measured the execution time for the case when a task fails to acquire semaphore instances and priority inheritance is carried out, occurring at point 3 on Fig. 5.

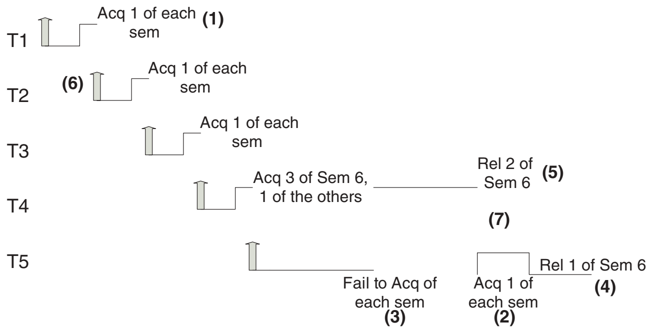


Fig. 5. Task execution of test program

The results show that the acceleration is greatest when a task successfully acquires semaphore units when the list is full: in this situation, the software implementation takes over 30 times more CPU cycles to execute than with the hardware accelerator. The gain is smallest when a task fails to acquire semaphore units; in this case priority inheritance requires considerable processing by the CPU that the accelerator cannot alleviate.

Release Function Results. The best case execution time of the software implementation takes place at point 4 on Fig. 5, when a task gives up all the units of a semaphore that it owns. The worst case scenario occurs when a task gives up only a fraction of the instances of a particular semaphore, occurring on point 5 of Fig. 5, since the list must be traversed to find the new location for the task/semaphore pair. The execution time with the hardware accelerator is still constant regardless of the state of the system. Table 1 shows the results of the *Release* operation. The hardware accelerated implementation executes more than 12 times faster than the software implementation in this situation.

Scheduler Function Results. The accelerator provides computational assistance when the CPU is running the scheduler function. Again, execution time depends on the state of the system: if the wait task remains blocked after the scheduler has finished, both implementations will take less time to execute because the task will not have acquired the semaphore units it is blocked on, and the task/semaphore list will not have to be

Table 1. CPU cycles to execute the Acquire and Release functionality and the scheduler function under different scenarios

Action	Implementation		Acceleration
	Software	Hardware	Gain
Acquire semaphore units (best case)	31472	5440	x 5.68
Acquire semaphore units(worst case)	165952	5440	x 30.5
Fail to acquire semaphore units and execute priority inheritance	48576	34944	x 1.39
Release semaphore unit (best case)	21504	4672	x 4.60
Release semaphore units (worst case)	58112	4672	x 12.44
Wait task stays blocked (best case)	4544	1344	x 3.38
Wait task unblocks (worst case)	40512	5763	x 7.03

updated, occurring at point 6 in Fig. 5. Conversely, if the waiting task becomes the running task, as seen at point 7 in Fig. 5, the execution time of the scheduler will be longer because semaphore units will have been acquired and the list will have been updated. Table 1 shows that the accelerator speeds up the scheduler up to 7 times.

8 Conclusion

In this paper, we have proposed a hardware accelerator for an access-control protocol for multiple-unit resources in a uniprocessor environment. Specifically, software and hardware implementations of the Priority Ceiling Protocol for multiple-unit resources were developed and performance numbers of the two implementations were compared. As expected, the hardware accelerator showed impressive gains over the software implementation. By using a high degree of parallelism to carry out otherwise time consuming computations, the hardware implementation executes PCP in predictable amounts of time. Future work may involve adapting the accelerator to a multi-processor system, allowing it to provide even more support to the underlying OS.

References

1. Akgul, B., Mooney, V., Thane, H., Kuacharoen, P.: Hardware Support for Priority Inheritance. In: Proceedings of the IEEE Real-Time Systems Symposium (RTSS'03) (2003) 246–254
2. Atmel Corporation, ATmega103L Datasheet. (2001) http://www.atmel.com
3. Belik, F.: An Efficient Deadlock Avoidance Technique. IEEE Transactions on Computers **39** (1990) 882–888
4. Cahit, i.: Deadlock Detection using (0, 1)-Labeling of Resource Allocation Graphs. Computers and Digital Techniques, IEE Proceedings **145** (1998) 68–72.
5. Gupta R., De Micheli, G.: Hardware/Software Co-Design. IEEE Proceedings **85** (1997) 349–365
6. Kohout, P., Ganesh, B., Jacob, B.: Hardware Support for Real-Time Operating Systems. In: Proc. First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2003), Newport Beach, CA (2003) 45–51
7. Labrosse, J.J.: MicroC/OS-II: The Real-Time Kernel. CMP Books (2002)
8. Liu, J.W.S.: Real Time Systems. Prentice Hall, New York, NY (2000)
9. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers **39** (1990) 1175–1185
10. Shiu, P.H., Yudong, T., Mooney, III, V.J.: A Novel Parallel Deadlock Detection Algorithm and Architecture. In: Hardware/Software Codesign, Proc. of 9th CODES (2001) 73–78
11. Wolf, W.: Computers as Components. Morgan Kaufman (2000)
12. Xilinx, http://www.xilinx.com.

Pattern Matching Acceleration for Network Intrusion Detection Systems[*]

Sunil Kim

School of Information and Computer Engineering, Hongik University,
72-1 Sangsu-Dong, Mapo-Gu, Seoul, Korea
sikim@cs.hongik.ac.kr

Abstract. Pattern matching is one of critical parts of Network Intrusion Detection Systems (NIDS). Pattern matching is computationally intensive. To handle an increasing number of attack signature patterns, a NIDS require a multi-pattern matching method that can meet the line-speed of packet transfer. The multi-pattern matching method should efficiently handle a large number of patterns with a wide range of pattern lengths and noncase-sensitive pattern matches. It should also be able to process multiple input characters in parallel. In this paper, we propose a multi-pattern matching hardware accelerator based on Shift-OR pattern matching algorithm. We evaluate the performance of the pattern matching accelerator under various assumptions. The performance evaluation shows that the pattern matching accelerator can be more than 80 times faster than the fastest software multi-pattern matching method used in Snort, a widely used open-source NIDS.

1 Introduction

In network security, pattern matching is extensively used inside Network Intrusion Detection Systems (NIDS) to detect signatures of unauthorized attempts to access and manipulate information and to render a system unreliable or unusable. A NIDS monitors both ingress and egress network traffic and compares the network packets with certain intrusion signatures or analyzes network traffic to find any suspicious anomalous/irregular pattern. A NIDS makes use of a set of rules that are applied to matching packets. A rule includes a signature of a malicious packet and an associated action to take if all conditions of the rule are met. A multi-pattern matching method is used to find a match of string patterns specified in rules against the content of a packet payload. Pattern matching is computationally intensive. The pattern matching routines in a widely used open-source NIDS, Snort account for up to 70% of total execution time and 80% of instructions executed on real traces [1].

The one of most challenging pattern matching problems for NIDS is the huge number of patterns to detect whose size ranges from one to more than one hundred bytes [2, 3]. The number of patterns will keep increasing, as more and more different attacks will continue to appear in the future. Another problem is that many of attack patterns

[*] This work was supported by 2005 Hongik University Research Fund.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 289–298, 2005.

are noncase-sensitive [3]. Therefore, pattern matching methods should handle noncase-sensitive pattern matching efficiently. The final challenge is that the pattern matching methods should process more than one incoming characters in parallel. By processing multiple input characters, we can easily boost the speed of pattern matching process for NIDS.

There have been many multi-pattern matching methods proposed. One kind of approaches is based on software implementations of pattern matching algorithms [2, 3, 4, 5, 6]. For a line speed of 10Gbps and beyond, most software approaches likely fail to meet the speed requirement. The other approaches are based on hardware implementations [7, 8, 9, 10, 11, 12, 13]. Some of them [8, 9, 10] implement regular expressions (NFAs/DFAs) using FPGA. Other approaches [11, 12, 13] use CAM (content-addressable memory) or comparator logic circuit. Both approaches require reprogramming of FPGA every time patterns are changed. In addition, incoming characters are globally broadcast to all character matchers or CAMs. This requires the use of an extensive pipelined broadcast tree to achieve a high clock rate. The operating frequency of these architectures tends to drop gradually as the number of patterns to handle is increased [13].

Another hardware approach implements blooming filters in which hash functions are used to find a pattern match [7]. A blooming filter is needed for each pattern size, and the same number of memory ports as the number of blooming filters is required to feed all blooming filters simultaneously. Therefore, as the number of pattern size is increased, the implementation becomes difficult. Unlike other hardware approaches mentioned above, this method does not require reprogramming of FPGA for patterns added. However, the blooming filter method could generate a false positive match, which requires the match rechecked by software.

In this paper, we propose a hardware pattern matching method. This method is based on a well-known single pattern matching algorithm, Shift-OR [14]. The algorithm is mainly performed by Shift and OR bit vector operations that can be efficiently implemented in hardware. The algorithm can be extended to support multi-pattern matching by concatenating all pattern vectors and process them all together. In general purpose processors, the algorithm is rather slow because the bit vector operation is limited by the word size. For a large number of patterns, we need many more iterations of Shift-OR operations for processing one input character. We propose a specialized pattern matching architecture that efficiently implements the multi-pattern Shift-OR algorithm. The architecture fully exploits current VLSI technologies that can allow high on-chip memory bandwidth and efficiently handle large-size bit vector operations. The architecture satisfies the domain specific pattern matching characteristics for NIDS: a large number of patterns with wide range of sizes, noncase-sensitive pattern matches and parallel processing of multiple input characters.

This paper begins by describing Shift-OR algorithm in Section 2. The detail of the proposed hardware pattern matching architecture will be described and evaluated in Section 3 and 4 respectively. We conclude in Section 5.

2 Shift-OR Pattern Matching Algorithm

In this section, we briefly describe Shift-OR pattern matching algorithm for a single pattern, which is the basis of the pattern matching architecture we present in this paper. The algorithm uses bitwise techniques. It keeps a bit array of size m (pattern length), a state vector R that shows if prefixes of the pattern match at the current place. For example, there are a pattern $P = p_0 \dots p_{m-1}$ and input string $X = \dots x_{i+j} \dots$. After processing x_{i+j}, $R[j] = 0$ if $x_i \dots x_{i+j}$ matches $p_0 \dots p_j$, otherwise $R[j] = 1$. There is another bit array of size m, a character position vector S_c, denoting the position of character c in pattern P. For example, $S_c[i] = 0$ if $p_i = c$, otherwise $S_c[i] = 1$. If we know that the bit value of $R[j]$ after processing x_{i+j}, we can easily compute $R[j+1]$ by knowing whether the next character x_{i+j+1} appear at pattern position p_{j+1}. $R[j+1]$ can be defined as follows:

$$R[j+1] = \begin{cases} 0 \text{ if } R[j] = 0 \text{ and } S_c[j+1] = 0 \text{ where } c = x_{i+j+1} \\ \\ 1 \text{ otherwise.} \end{cases} \tag{1}$$

$$R[0] = S_c[0] \text{ where } c = x_{i+j+1} \tag{2}$$

$R[m-1] = 0$ means the pattern $x_i \dots x_{i+m-1}$ matches $p_0 \dots p_{m-1}$, that is, the matching pattern is found. The computation of new R for the next input character c reduces to Shift and OR operations ($SHIFT(R)$ OR S_c).

This algorithm easily handles any finite class of symbols, complement symbols and even don't care symbols. If position i of a pattern allows a class of symbols $\{$x,y,z$\}$, then letting $S_x[i] = S_y[i] = S_z[i] = 0$ handles the case. Complement symbols and don't care symbols can be handled in the same way. Therefore, noncase-sensitive matches can be easily processed without any additional overhead. The algorithm can be extended for multiple patterns. It first coalesces all state vector R_s for each pattern into one state vector. It also coalesces all character position vector S_cs for each pattern into one character position vector for a given character c. The only difference from single pattern match is that when the new bit value of R corresponding to the first position of a pattern, i, is computed, the value is only affected by $S_c[i]$, not by shifted value from $i-1$th position.

3 Hardware Pattern Matching Acceleration for NIDS

This pattern matching hardware accelerator (PMA) consists of four components: character position vector unit, state vector computation unit, match detection unit, and priority encoder unit. The character position vector unit takes N input characters from the input buffer and generates N character position vectors, one for each input character. The input buffer contains the payload of a packet to examine. N is the shift size of the accelerator, that is, N input characters are processed in parallel. Figure 1 shows the architecture of the first component, character position vector unit. There are N character position vector tables. Each table has 256 character position vectors, one per an 8-bit character. The character position vector is a coalesced character position vector for all

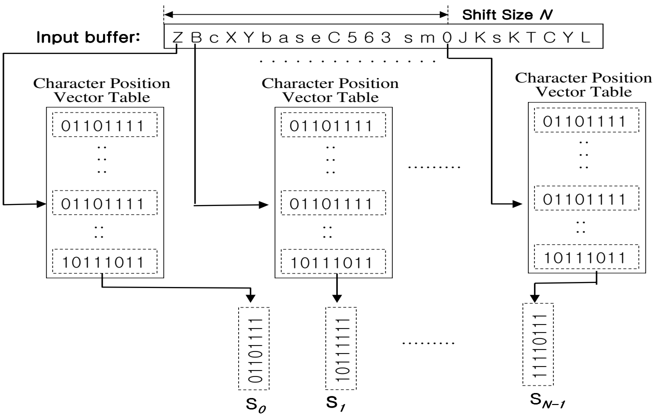

Shift Size N
Input buffer:
Z B c X Y b a s e C 5 6 3 s m 0 J K s K T C Y L
Character Position Vector Table
01101111
01101111
10111011
Character Position Vector Table
01101111
01101111
10111011
Character Position Vector Table
01101111
01101111
10111011
01101111
10111111
11110111
S0
S1
SN-1

Fig. 1. Character position vector unit

the patterns for a given character. These vectors are precomputed from string patterns and loaded into the table. All the tables have the same character position vectors. Each input character is used as an index to the corresponding table to generate a character position vector S_i as shown in Figure 1.

The N character position vectors are fed into the next component that computes state vector R. The state vector computation unit takes those character position vectors, state vector R generated from the previous cycle and pattern boundary vector B and then computes a new state vector and stores it into R. The pattern boundary vector B denotes boundaries of each pattern by bit value 0 in a vector of the same size of R. Figure 2 shows the execution of the state vector computation unit. Initially R is *AND*ed with B, then shifted and *OR*ed with S_0 to generate intermediate state vector T_0. Next, T_0 is *AND*ed with B, then shifted and *OR*ed with S_1 to generate next intermediate state vector T_1. The same computation is performed at each stage until T_{N-1} is generated. The final result T_{N-1} will be stored into R again for the next cycle computation. The computation is represented in the following equations.

$$T_k(0) = S_k(0) + 0 = S_k(0) \qquad \text{for all } k \tag{3}$$

$$T_0(i) = S_0(i) + (R(i-1) * B(i-1)) \qquad \text{for } i > 0 \tag{4}$$

$$T_k(i) = S_k(i) + (T_{k-1}(i-1) * B(i-1)) \quad \text{for } k > 0, i > 0 \tag{5}$$

The AND operations with B prevent the computation result from propagating cross pattern boundaries. As shown in Figure 2, the shift operations are performed by simply connecting the ith position results to one input port of the OR gate of the $i+1$th position at the next stage. Each stage computation is equivalent to one Shift-OR operation in Shift-OR algorithm. The state vector computation unit can perform N Shift-OR operations in a single cycle. Combinatorial logic circuit is used for all the computation, and intermediate state vectors, $T_0 \ldots T_{N-1}$, are generated on the fly and does not need to be stored.

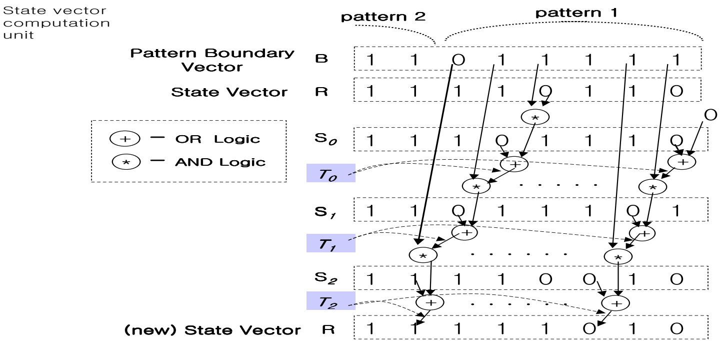


Fig. 2. State vector computation unit

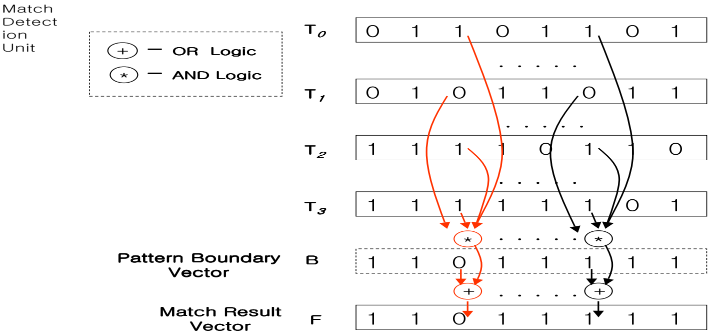


Fig. 3. Matching detection unit

In the middle of *N* input characters being processed, matches can be found. The third component of this accelerator, match detection unit, detects all the matches from all the intermediate state vectors. Figure 3 shows the architecture. All the bits at the same position in the intermediate state vectors are *AND*ed and then *OR*ed with the bit at the same position in the pattern boundary vector *B*. The result is stored into the match result vector *F*. If the match result vector has zero bits (*match bits*), it means there are matches. The fourth component of the accelerator, priority encoder logic, computes the index of the first match bit position from the match result vector *F*. The match bit position information is used by software to find the pattern that matches. Other detailed design issues for interactions between the accelerator and the software are not addressed because these are not main concerns of this paper.

4 Performance Model and Evaluation

We evaluate the performance of the proposed pattern matching hardware accelerator (PMA) using a simulation model and compare it to the performance of software multi-pattern matching methods employed in Snort 2.1.2, *AC* and *MWM*, which are based on Aho-Corasick [5] and Wu-Manber [6] algorithms, respectively. The implementation details of these multi-pattern string matching algorithm used in Snort can be found in [2].

We assume that PMA is used in Snort in the place of the software pattern matching methods. The performance of PMA is evaluated based on the execution data of software PMA gathered while executing Snort. The software PMA simulates the execution of the pattern matching accelerator described in the previous section. Only pattern matching operations are evaluated. Other operations related to PMA setup, such as loading the character position vector table and the pattern boundary vector, are excluded because such operations are not time-critical.

4.1 PMA Execution and Performance Model

We first define PMA *pattern matching operation* and analyze their execution time. During each pattern matching operation, N characters (*shift size*) are read from the input buffer and the character position vector tables are accessed, N intermediate state vector are computed, and finally the match result vector is generated from the pattern boundary vector B and the N intermediate state vectors. The execution time for pattern matching operations can vary depending on implementation technology. For evaluation, we explore a wide range of execution times for the pattern matching operation. The execution time is measured in number of cycles.

When a pattern match starts, all the internal vectors including the state vector and the matching result vector are reset to their initial condition. After that, pattern matching operations repeat until a match is found, and the match result vector records all the matches found. When matches are found, an interfacing software finds the first match bit position from the match result vector by reading the output of the priority encode logic. The match bit is cleared after its position is read. If there is more than one match bit, the position of the next match bit is computed by the priority encoder logic for the next use. This operation continues until all the match bit positions are read and cleared. The software can find the index of the matching pattern from the match bit position and performs other operations needed to conclude a rule match. This whole pattern matching operation will continue until all the characters in the input buffer are processed.

The execution time of PMA depends on the number of pattern matching operations and the total execution time of the priority encoder logic. We first assume that reseting all internal vectors takes one cycle. We largely divide the pattern matching operation into two parts: *memory access* and *vector computation*. The memory access time(T_m) is the time taken for accessing the input buffer and the character position vector tables. If the input buffer and the character position vector table are implemented with the same static RAM technology as that of the first level cache, the access time could be as fast as one cycle for each input buffer and character position table. If a slower memory technology is used, the access time could be more than a hundred cycles. For evaluation, we vary the memory access time from 2 to 64 cycles.

The vector computation consists of N Shift-OR operations to compute intermediate state vectors and one match detection. The Shift-OR takes two gate delays (one AND and one OR gates), and the match detection takes one or four gate delays (zero to four AND gates and one OR gates) depending on the number of the intermediate state vectors. The Shift-OR operation can be very fast. We vary the execution time of the Shift-OR operation (T_{so}) from 0.25 to 1 cycle and assume that the match detection takes 2 cycles. We use the roundup value of the vector computation cycles in case that it has a fraction value.

The total execution time of the priority encoder logic depends on the number of matches. This priority encoder logic has a large fan-in. The fan-in size is the sum of all pattern size M, which is about 24K for default rules that come with Snort 2.1.2. Such a large priority encoder can be implemented using Parallel Priority look-ahead architecture [15]. The gate delay is approximately $\log_2 M - 3$, which is about 12 gate delays. For evaluation, we vary the execution time of the priority encoder (T_p) from 2 to 8 cycles. The total execution cycles for PMA to process one payload can be expressed as follows.

$$\begin{aligned} \text{PMA execution time} \;=\;& 1 + (T_m + \lfloor N \times T_{so} + 2 \rfloor) \times N_{po} + T_p \times N_{pm} \\ \\ \text{where } N_{po} \;& \text{is the number of pattern matching operations} \\ N_{pm} \;& \text{is the number of pattern matches} \\ N \;& \text{is shift size} \\ T_m \;& \text{is the memory access time} \\ T_{so} \;& \text{is the execution time of Shift-OR operations} \\ T_p \;& \text{is the execution time of priority encoder logic} \end{aligned} \tag{6}$$

In the above equation, '1' is for reseting all internal vectors. The second term is for the execution time for all the pattern matching operations performed, and the third term is for the total execution time of the priority encoder logic. We obtain the information such as the number of pattern matching operations(N_{po}) and the number of pattern matches(N_{pm}) from Snort runs on packet traces and evaluate the execution time for PMA using the equation 6 and varying N, T_m, and T_p as explained above. We use the packet trace captured during the Capture the flag contest at DefCon 11 [16]. Defcon's Capture the Flag(*CtF*) game is the largest open computer security hacking game.

4.2 Performance Analysis

The execution time for the software multi-pattern match methods are measured in cycles by running Snort on a Linux PC with 2.4GHz Pentium IV and 512MB memory. We use a Time-stamp counter [17] and count the number of cycles to run the pattern match softwares. The Figure 4 shows the speedup of PMA hardware and the software multi-pattern matching methods with respect to the execution time of *MWM*. The label PMA(*X,Y,Z*) in X-axis denotes the PMA with $T_m = X, T_{so} = Y$, and $T_p = Z$ cycles. The result shows that *MWM* is about 2 times faster than *AC* method. The result also shows the memory access time T_m affects the performance of PMA significantly more than the priority encoder logic execution time T_p does. The increase in priority encoder logic execution time from 1 to 32 cycles does not change the speedup of PMA much, whereas the increase in the memory time from 2 to 64 cycles dramatically reduces the speedup of PMA. This is because the number of pattern matching operations N_{po} is much larger

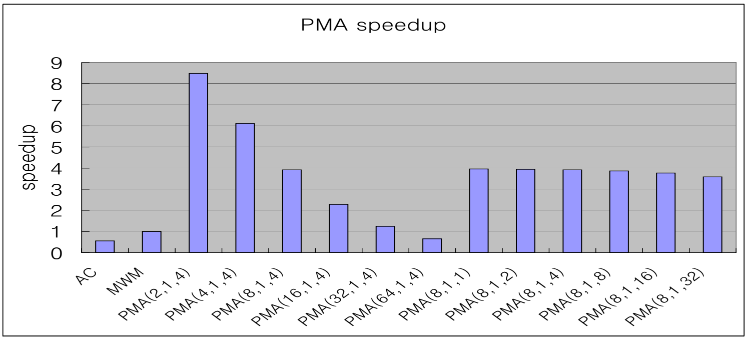


Fig. 4. PMA speedup with varying memory access time and priority encoder execution time (*Shift size* = 1)

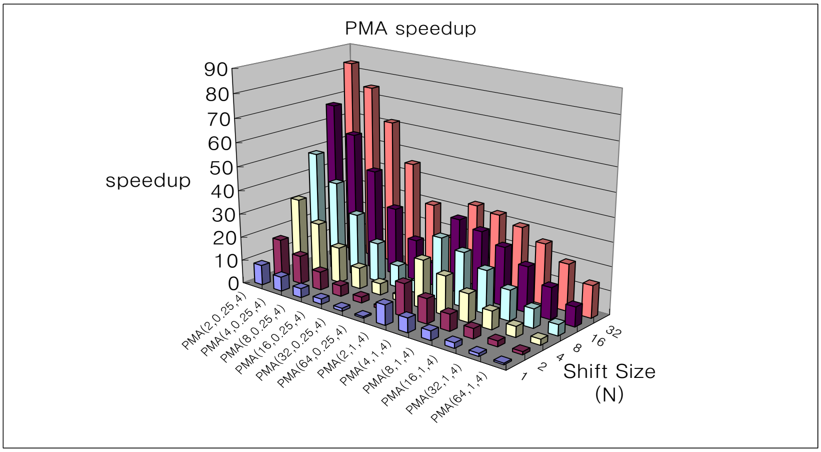


Fig. 5. PMA speedup with varying shift size *N*, memory access time and Shift-OR operation time

than the number of pattern matches N_{pm}. When the execution time of Shift-OR operation and the priority encoder logic is fixed to 1 and 4 cycles respectively, and memory access time is increased from 2 to 64 cycles, PMA speedup drop from 8 to 0.5. This shows the performance of PMA is sensitive to the memory access time.

We also evaluate PMA speed up varying the shift size *N* from 1 to 32 and Shift-OR operation time from 0.25 to 1 cycle. Figure 5 shows that increasing shift size improves the performance of PMA almost linearly. The effect of a large shift size becomes more significant when the Shift-OR operation time is smaller. This implies that we need to have a large shift size to fully exploit the fast Shift-OR operation time. The result shows that when the memory time takes 2 cycle and Shift-OR operations takes a quarter cycle, PMA could be more than 80 times faster than *MWM* with 32 character position tables and about 70 times faster with 16 character position tables. This result shows that the fast memory access time and a large shift size of PMA is very important to achieve a good performance of PMA.

The shift size is limited by the data read width of the input buffer and chip areas to accommodate the character position vector tables. The current VLSI technology can satisfy the implementation requirements for high speed PMA. The memory access time can be optimized by either using fast memory technologies or pipelining the access path of the input buffer and character position tables. Memory areas for character position tables for large shift size N can also be accommodated with current VLSI technology. For example, Intel is introducing Itanium processor with 26.5MB on-chip cache [18]. Given that each character position vector table takes 768K (256 x 24K bits) bytes, 32 tables (24Mbytes) can be built with the VLSI technology. We can also partition the architecture into multiple chips if chip area becomes a problem.

5 Conclusions

Pattern matching is one of critical parts of Network Intrusion Detection Systems (NIDS). Pattern matching for NIDS is computationally intensive and need to handle a large number of patterns with a wide range of pattern lengths, and noncase-sensitive pattern matches. It also needs to process multiple input characters in parallel.

We proposed a specialized hardware multi-pattern matching architecture based on Shift-OR algorithm. The proposed pattern matching architecture efficiently satisfies all the requirements of pattern matching in NIDS. We evaluated the performance of the hardware pattern matching architecture in a wide range of timing assumptions and found that the pattern matching architecture could be more than 80 times faster than the fastest software pattern matching method used in the current Snort.

References

1. Antonatos, S., Anagnostakis, K.G., Markatos, E.P.: Generating realistic workloads for network intrusion detection systems. In: Proceedings of ACM Workshop on Software and Performance. (2004)
2. Tuck, N., Sherwood, T., Calder, B., Varghese, G.: Deterministic memory-efficient string matching algorithms for intrusion detection. In: Proceedings of the 23rd Conference of the IEEE Communication Society (INFOCOM04). (2004)
3. Liu, R., Huang, N., Chen, C., Kao, C.: A fast string matching algorithm for network processor based intrusion detection system. ACM Transaction on Embedded Computing Systems **3** (2004) 614–633
4. Markatos, E.P., Antonatos, S., Polychronakis, M., Anagnostakis, K.G.: Exclusion-based signature matching for intrusion detection. In: Proceedings of the IASTED International Conference on Communications and Computer Networks. (2002)
5. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Communications of the ACM **18** (1975) 333–340
6. Wu, S., Manber, U.: AGREP - a fast approximate pattern-matching tool. In: Proceedings of the 1992 Winter USENIX Conference, San Francisco, CA (1992)
7. Dharmapurikar, S., Krishnamurthy, P., Sproull, T.S., Lockwood, J.W.: Deep packet inspection using parallel bloom filters. IEEE Micro **24** (2004)

8. Hutchings, B.L., Franklin, R., Carver, D.: Assisting network intrusion detection with reconfigurable hardware. In: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. (2002)
9. Sidhu, R., Prasanna, V.K.: Fast regular expression matching using FPGAs. In: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines. (2001)
10. Moscola, J., Lockwood, J., Loui, R.P., Pachos, M.: Implementation of a content-scanning module for an internet firewall. In: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines. (2003)
11. Gokhale, M., Dubois, D., Dubois, A., Boorman, M., Poole, S., Hogsett, V.: Granidt: Towards Gigabit rate network intrusion detection technology. In: Proceedings of the 12th International Conference on Field-Programmable Logic and Applications. (2002)
12. Cho, Y.H., Navab, S., Mangione-Smith, W.H.: Specialized hardware for deep network packet filtering. In: Proceedings of the Field Programmable Logic and Applications. (2002)
13. Sourdis, I., Pnevmatikatos, D.: Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In: Proceeding of the 12th Annual IEEE Symposium on Field Programmable Custom Computing Machines. (2004)
14. Baeza-Yates, R.A., Gonnet, G.H.: A new approach to text searching. In: Proceedings of ACM 12th International Conference on Research and Development in Information Retrieval. (1989)
15. Kun, C., Quan, S., Mason, A.: A power-optimized 64-bit priority encoder utilizing parallel priority look-ahead. In: Proceedings of the IEEE Int. Symposium on Circuits and Systems. Volume 2. (2004) 753–756
16. The Shmoo Group: Capture the RootFu! http://www.shmoo.com/cctf/ (2005)
17. Intel Corp.: IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. (2004)
18. Naffziger, S., Grutkowksi, T., Stackhouse, B.: The implementation of a 2-core multi-threaded Itanium family processor. In: Proceedings of Solid-State Circuits Conference. (2005)

Real-Time Stereo Vision on a Reconfigurable System

SungHwan Lee, Jongsu Yi, and JunSeong Kim

School of Electrical and Electronics Engineering, Chung-Ang University,
221 HeukSeok-Dong DongJak-Gu, Seoul, Korea 156-756
{lshcau, xmxm2718}@wm.cau.ac.kr, junkim@cau.ac.kr

Abstract. Real-time three-dimensional vision would support various applications including a passive system for collision avoidance. It is a good alternative of active systems, which are subject to interference in noisy environments. In this paper, we investigate the optimization of real-time stereo vision with respect to resource usage. Correlation techniques using a simple sum of absolute differences(SAD) is popular having good performance. However, processing even a small image takes seconds. In order to provide depth maps at frame rate around 30fps, which typical cameras can provide, hardware accelerations are necessary. Regular structures, linear data flow and abundant parallelism make the correlation algorithm a good candidate for reconfigurable hardware. We implemented versions of SAD algorithms in VHDL and synthesized them to determine resource requirements and performance. By decomposing a SAD correlator into column SAD calculator and row SAD calculator with buffers in between we showed around 50% savings in resource usage. By altering the shape of correlation windows we found that a 'short and wide' rectangular window reduced storage requirements without sacrificing quality compared to a square one.

1 Introduction

A collision avoidance system for a device with mobility requires the ability to build a three-dimensional map of its environment. Traditionally, this has been accomplished by active sensors, which send a pulse - either electromagnetic or sonar - and detect the reflected return[1, 2]. Such active systems work well in the environments with small number of moving devices and thus the probability that active sensors will interfere is low. However, when the density of moving objects becomes high, active systems are easily left in noisy environments. Lots of moving objects with a wide range of speeds and directions create that many reflections at various strengths. A sensor could easily be confused by extremely weak reflections of its own and strong pulses from other objects. Passive systems, on the other hand, are much less sensitive to environmental interference. Stereo vision is one of the representative passive systems. Typical cameras can provide 30 or more images per second and each pair of images can provide a complete three-dimensional map of the environment. However, processing even small low resolution images takes more than a second in software. This is well below the frame rates obtainable with commodity cameras and may be far too slow to enable even relatively slow moving objects to avoid colliding each other. Thus, hardware accelerators are required in order to obtain real-time 3D environment maps. Software simulations

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 299–307, 2005.

have determined that correlation techniques using a simple sum of absolute differences (SAD) algorithm perform well [3,4].

In our previous work [5], we showed that accurate real-time three-dimensional maps are feasible with modern FPGA technology. A SAD correlator with its associated accuracy and speed requirements could be fitted onto a single commercially available FPGA. In this paper, we present versions of optimization of the SAD correlator with respect to resource usage. By decomposing the original SAD correlator into column SAD calculator and row SAD calculator with buffers in between we reduced the number of adders from the original SAD correlator. Also, by utilizing rectangular windows instead of traditional square windows we saved more resources without sacrificing accuracy. In the remainder of the paper, Section 2 briefly surveys the stereo image matching techniques, Section 3 provides the concept of the SAD algorithm and Section 4 introduces our SAD correlators. Section 5 then provides the results of our experiments. Finally, Section 6 summarizes our results and conclusions.

2 Stereo Image Matching

Stereo vision refers the problem of extracting 3-dimensional structure from two(or more) images taken from different viewpoints[6]. Image matching is an important part in stereo vision system involving two main problems: *correspondence* and *reconstruction*. The correspondence problem consists of determining, given a pair of stereo images, which parts in the left(right) image correspond to which parts in the right(left) image. Since there are parts of a scene projected on a single image only it must be able to tell the parts in each image that should not be matched. The reconstruction problem consists of determining, given a set of corresponding parts of a pair of stereo images, 3-dimensional location and structure of the observed objects.

Ideally, we want to find all matching pixels of a pair of stereo images. However, the value of brightness of a single pixel is too low to determine its correspondence. Instead, sets of pixels are used for real stereo matching algorithms. Correlation-based methods and feature-based methods are the two representation stereo matching classification [6,7].

In correlation-based methods, image windows, arrays of neighboring pixels, of fixed size are used. Given a pair of stereo images, one window is fixed in the left(right) image and the other window is moving in the right(left) image. By comparing the windows from the pair of images correlation is measured and the window, that maximizes the similarity criterion, is determined. Normalized Cross-Correlation(NCC), Sum of Squared Differences(SSD), Sum of Absolute Differences(SAD), Census, and Rank algorithms are popular matching metrics[7,8].

Feature-based methods use a sparse set of features instead of image windows. These include occlusion edges, vertices of linear structures, prominent surface markings, zero crossings and patches by the Moravec operator[9]. Corresponding elements from a pair of stereo images are given by the most similar feature pair, the one associated to the minimum distance. Feature-based methods cannot detect small changes in stereo images and is not suitable when images have no boundary.

3 SAD Algorithm

Area-based correlation algorithms attempt to find the best match between a window of pixels in one image and a window in the other image[6]. The matching process is illustrated in Figure 1. The window centerd on pixel P in the left image is moved through the disparity range until the best match is found with a windwos centerd at P in the right image. aligning the two cameras to meet the epipdar constraint ensures that P must lie on the same scan line in each image [6, 7].

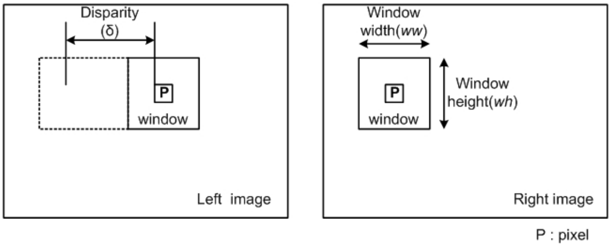


Fig. 1. Correlation based matching

In the SAD algorithm, the criterion for the best match is minimization of the sum of the absolute differences of corresponding pixels in a window. The correlation algorithm has a regular structure and simple data flow making it good for implementation in reconfigurable hardware. The SAD function is defined to be

$$C(x,y,\delta) = \sum_{y=0}^{wh-1} \sum_{x=0}^{ww-1} I_R(x,y) - I_L(x+\delta,y) \tag{1}$$

The SAD function $C(x,y,\delta)$ is evaluated for all possible values of the disparity, δ, and the minimum is chosen. In the equation, $I_R()$ and $I_L()$ mean right image and left image respectively. The x, y represent coordinates in pixel in a single image, ww and wh represent window width and height, and δ represents disparity number. For parallel camera axes, δ ranges from 0 for objects at infinity to Δ for objects at the closest possible to the camera. The correlation algorithm has regular structures having abundant parallelism - $C(x,y,\delta)$ can be evaluated in parallel for each $\delta \in [0,\Delta]$. The SAD function requires only adders and comparators for which modern FPGAs provide good supports. However, accurate depth maps require large disparity ranges and high resolution images - both of which provide challenges to fitting a full correlator on a single FPGA.

4 SAD Correlator

We have implemented versions of the SAD correlation algorithm in VHDL and synthesized them to determine their resource requirements and performance. Each version completely accomplishes the SAD correlation algorithm and has the following features:

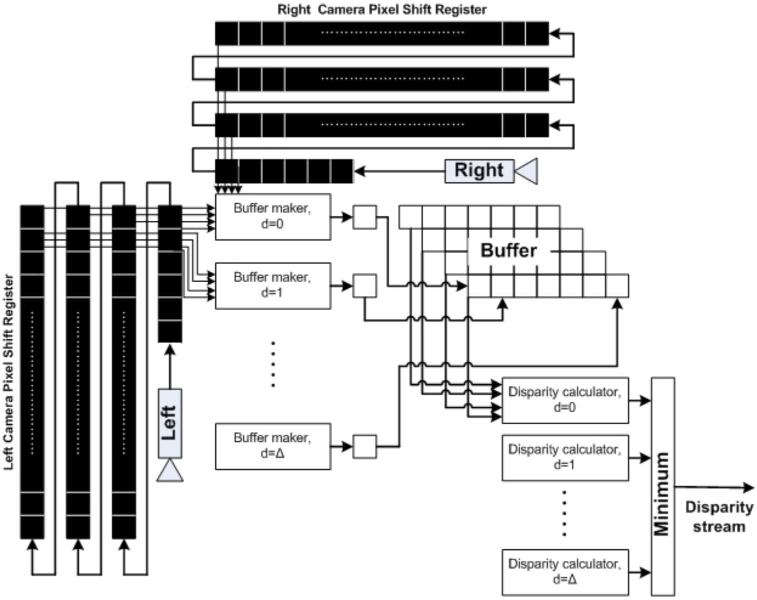

Right Camera Pixel Shift Register
Right
Left Camera Pixel Shift Register
Left
Buffer maker, d=0
Buffer maker, d=1
Buffer maker, d=Δ
Buffer
Disparity calculator, d=0
Disparity calculator, d=1
Disparity calculator, d=Δ
Minimum
Disparity stream

Fig. 2. Block diagram for the SAD correlator v1.2

Table 1. SAD Correlator Resource requirements

SAD correlator	Object	Count	Size
shift register	register	2	$sl \times (wh - 1) + \Delta + 1$
buffer maker	subtractor	$\Delta + 1$	wh
	adder	$\Delta + 1$	$wh - 1$
buffer	buffer	1	$ww \times (\Delta + 1)$
disparity calculator	adder	$\Delta + 1$	$ww - 1$
minimum detector	comparator	1	Δ

- SAD correlator v1.0[5]
  fully implements the SAD correlation algorithm without any optimization
- SAD correlator v1.1
  decomposes SAD correlator into column SAD calculator (buffer maker) and row SAD calculator (disparity calculator) placing buffers in between reducing the number of adders from the SAD correlator v1.0
- SAD correlator v1.2
  does a certain approximation in SAD calculation by ignoring the least significant bit (reducing the number of bits in adders): at the cost of accuracy fo further save space from the SAD correlator v1.1

A block diagram of the SAD correlator v1.2 is shown in Figure 2. Pixels stream in from both cameras into the long left and right shift registers, which store sufficient pixels so that all the pixels in a correlation window are available to the buffer maker at the same time. The key parameters determining the size and performance of an SAD correlator are ① the scan line length, *sl*, ② the window width, *ww*, ③ the window height, *wh*, and ④ the maximum disparity, Δ.

Basic resource requirements are indicated in Table 1. To a first approximation, the resource requirements for an SAD correlator are given by:

$$\begin{aligned} cost_{SAD} \approx\ & 2 \times (sl \times (wh-1) + \Delta + 1) \times c_{reg} && (shift\ register) \\ & +(\Delta+1) \times (wh \times c_{AD} + (wh-1) \times c_{sum} && (buffer\ maker) \\ & +ww \times (\Delta+1) \times c_{reg} && (buffer) \\ & +(\Delta+1) \times (ww-1) \times c_{sum} && (disparity\ calculator) \\ & +\Delta \times c_{comp} && (comparator) \\ & +c_{overheads} && (control,\ etc.) \end{aligned} \tag{2}$$

Where c_{AD} is the cost of absolute difference circuit, c_{sum} is the cost of an adder, c_{comp} is the cost of a comparator, c_{reg} is the cost of a pixel register, $c_{overheads}$ is the cost of control and steering logic. This relation should be a good predictor for low values of all the application parameters, where all overheads can be lumped effectively into the single overheads term. Key contributors to the delay of the correlator are from the $(wh-1)$ adders in buffer makers and the $(ww-1)$ adders in disparity calculators. A simple VHDL model which performs the additions in a loop adds a delay of $O(wh+ww-2)$ to the circuit. However, for better performance we use a tree adder, which costs delay of $O(log(wh+ww-2))$. Note that the synthesizer is able to produce a compact circuit with the tree adder using the '+' operator, despite the triangular shape of the tree.

5 Experimental Results

For this experiment we use Xilinx Virtex-II XC2V8000 FPGAs[10] with scan line length, $sl = 320$ and the maximum disparity, $\Delta = 32$ (accuracy depends on the disparity value, so we ran trials to determine the value of Δ). Figure 3 shows the hardware

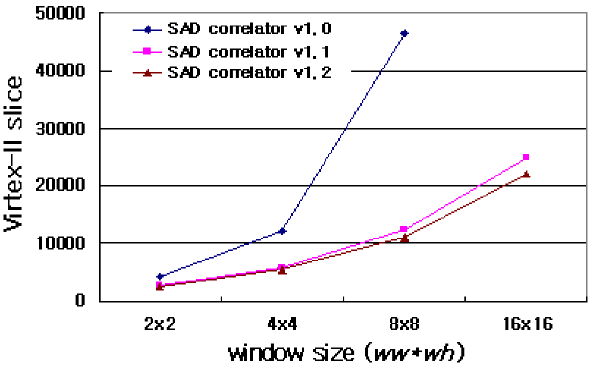
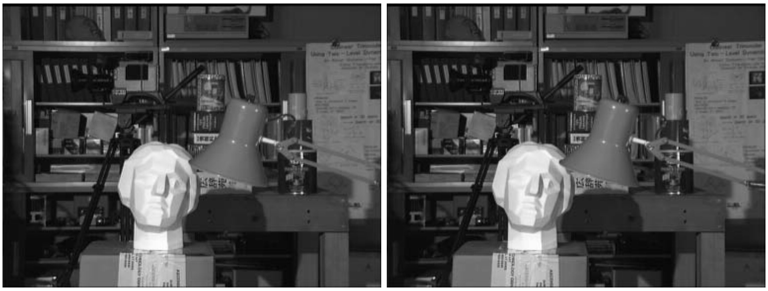


Fig. 3. Resource usage vs. window size $(ww \times wh)$ for versions of the SAD correlator

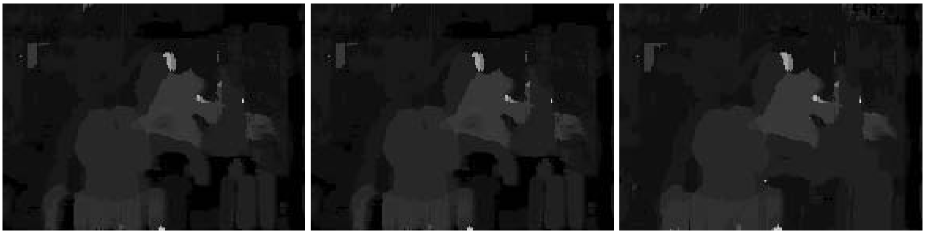
resource usage for the SAD correlators with various window sizes. The X-axis represents the square window sizes used in the experiment and Y-axis is the number of slices occupied.



(a) original left-image (b) original right-image

Fig. 4. A pair of Tsukuba images for test inputs

Figure 5 shows samples of resulting depth maps of versions of SAD correlator with windows size of $8 \times 8 (ww \times wh)$. A pair of the input images in figure 4 - Tsukuba image [3, 11] - consisting of 384×288 pixels was used as a test inputs. From the figure 5, we can see that there is little difference in depth map among the versions of SAD correlator.



(a) SAD correlator v1.0 (b) SAD correlator v1.1 (c) SAD correlator v1.2

Fig. 5. Sample depth maps of versions of the SAD correlator with 8×8 window

Figure 6 shows the simulation waveform using the Tsukuba image. The process time of one frame image is less than 120,000 c.c. The table 2 summarizes the performance of the SAD correlator v1.0 with various test images including the Tsukuba images, when it works in 10 MHz.

It is unnecessarily common to use square matching windows in correlation-style stereo algorithms. If you carefully look at the algorithm you can see that matching

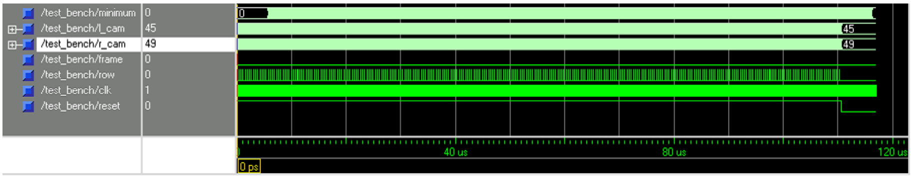
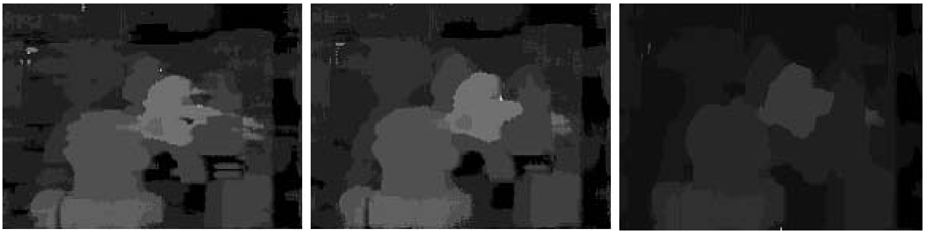


Fig. 6. The simulation waveform using the Tsukuba images

Table 2. The performance of the SAD correlator v1.0 in various environments

Image size (pixel)	maximum disparity (Δ)	window size ($ww \times wh$)	rate (frame/sec)
640×480	64	16×16	31
640×480	64	32×32	30
320×240	64	16×16	122
320×240	64	32×32	115

process only uses a small part of each scan line at any time - specifically, ww from the left image and $ww + \Delta + 1$ from the right image. The remaining pixels are stored in shift registers for use in subsequent cycles. Pixels in surrounding scan lines are only used to support matching by reducing noise effects. Figure 7 shows samples of depth maps of SAD correlator v1.2 for the same Tsukuba input images with various rectangular windows. We can find that a 'short and wide $(wh < ww)$' window produces similar matching quality to the square one. However, We can see that a considerable amount of space can be saved in an FPGA by using rectangular $(wh < ww)$ windows. Figure 8 shows a FPGA resource usage for various rectangular window sizes as well as square ones. We can conclude that a rectangular window in SAD correlators is worth utilizing, especially when ww is sufficiently large, since it saves lots of space (nearly 50% profit) without sacrificing quality.



(a) 16×4 window (b) 16×8 window (c) 16×16 window

Fig. 7. Sample depth maps of the SAD correlator v1.2 with various window sizes $(ww \times wh)$

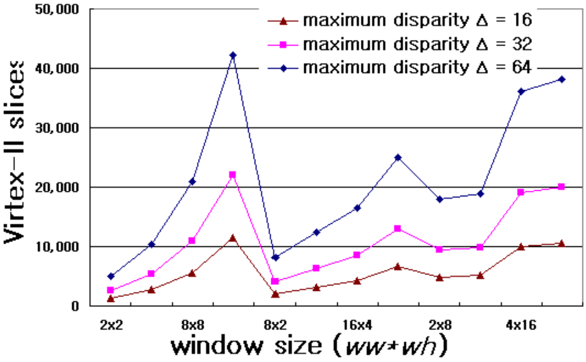


Fig. 8. Resource usage vs. window size $(ww \times wh)$ for various disparities

6 Conclusion

Accurate real-time 3D depth maps are feasible with modern FPGA technology. While the feasibility of a proposed application can be testified, in principle, by simply counting circuit elements needed to implement a module and using those counts in equation 2, place and route tools have to work from high level models and may have problems allocating and laying out circuits that a human engineer may not. FPGA implementations are also constrained by availability of routing resources and this factor is much harder to estimate than logic cell needs, thus practical trials of the type we carried out here are counts to determine real cost factors.

Decomposing a SAD correlator into column SAD calculator and row SAD calculator with buffers in between reduces number of adders: easily saving around 50% in resource usage. Simulation results show that altering the shape of the correlation window can further reduce the number of cells needed for inactive parts of scan lines. The SAD correlator v1.2 described in this paper will provide real-time performance at pixel clock rates up to $\sim$ 10 MHz.

Acknowledgements

This research was partially supported by the MIC(Ministry of Information and Communication), Korea, under the Chung-Ang University HNRC-ITRC(Home Network Research Center) support program supervised by the IITA(Institute of Information Technology Assessment).

References

1. Olson, C.F.: Maximum-likelihood image matching. IEEE Trans. Pattern Anal. Mach. Intell. **24** (2002) 853–857
2. Sebe, N., Lew, M.S.: Maximum likelihood stereo matching. In: ICPR '00: Proceedings of the International Conference on Pattern Recognition (ICPR'00)-Volume 1, Washington, DC, USA, IEEE Computer Society (2000) 1900

3. Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. Int. J. Comput. Vision **47** (2002) 7–42
4. Leclercq, P., Morris, J.: Assessing stereo algorithm accuracy. In: IVCNZ '02: Proceedings of Image and Vision Computing'02, University of Auckland, Auckland, New Zealand (2002) 89–93
5. Yi, J., Kim, J., Li, L., Morris, J., Lee, G., Leclercq, P.: Real-time three dimensional vision. In Yew, P.C., Xue, J., eds.: Asia-Pacific Computer Systems Architecture Conference. Volume 3189 of Lecture Notes in Computer Science., Springer (2004) 309–320
6. Barnard, S.T., Fischler, M.A.: Computational stereo. ACM Comput. Surv. **14** (1982) 553–572
7. Brown, M.Z., Burschka, D., Hager, G.D.: Advances in computational stereo. IEEE Trans. Pattern Anal. Mach. Intell. **25** (2003) 993–1008
8. Leclercq, P., Morris, J.: Robustness to noise of stereo matching. In: ICIAP '03: Proceedings of the 12th International Conference on Image Analysis and Processing, Washington, DC, USA, IEEE Computer Society (2003) 606
9. Grimson, E.L.: Computatoinal experiments with a feature based stereo algorithm. IEEE Trans. Pattern Anal. Mach. Intell. **PAMI-7** (1985) 17–33
10. http://www.xilinx.com: Virtex- platform fpgas : Complete data sheet (2002)
11. http://www.middlebury.edu/stereo: Middlebury stereo vision page (2005)

Application of Very Fast Simulated Reannealing (VFSR) to Low Power Design

Ali Manzak and Huseyin Goksu

Suleyman Demirel University, Isparta, Turkey
manzak@mmf.sdu.edu.tr, goksu@sdu.edu.tr

Abstract. This paper addresses the problem of optimal supply and threshold voltage selection with device sizing by minimizing power consumption and maximizing battery charge capacitance using Very Fast Simulated Reannealing (VFSR). We assume that multiple supply voltages and multiple threshold voltage devices are available at gate level. Minimizing power consumption does not necessarily maximize battery charge capacitance. This paper achieves this by implementing both objectives in the cost function.

1 Introduction

Reducing supply voltage is an effective way of decreasing power consumption. Dynamic power is related quadratically and leakage power is related almost linearly to supply voltage. Dynamic power still dominates the total power consumption of digital circuits which might change in near future. Automatic design tools generate digital circuits with predetermined supply and threshold voltages. These values might not be optimal in terms of power consumption and battery capacitance. New power optimization design tools might be necessary for power constrained application specific designs.

Power can be traded-off with delay and area using power optimization methods. The speed of a circuit is determined by the critical path. Designers minimize critical path delay with sizing. Since all the paths of circuit are not balanced, extra slack is available on non-critical paths. This extra slack can be used to minimize power consumption. Switching activity is also a determining factor since it varies with application software. Average switching activity of the circuit needs to be estimated for optimal performance.

Decreasing supply voltage is the most effective way of reducing power, since dynamic power is quadratically and leakage power is almost linearly proportional to supply voltage. Decreasing supply voltage however increases the circuit delay and decreases the throughput. Lowering threshold voltage increases circuit speed and helps circuit to satisfy the required frequency. On the other hand, decreasing the threshold voltage increases the leakage current exponentially. Therefore there is an optimum point for supply voltage and threshold voltage where delay constraint is satisfied and power and battery capacitance is optimum.

When gate sizing is added as another optimization parameter, circuit optimization gets more complex. Reducing supply voltage, using low-V_{th} transistors or decreasing transistor width reduces power consumption with the expense of an increase in delay. Using high voltage, low V_{th}, and large channel width for the transistors on the critical

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 308–313, 2005.

path, thus satisfying timing constraints and using low voltage, high V_{th}, and small channel width transistors on the non-critical path, thus minimizing power is a well-known power optimization method. However, optimal utilizition of extra slack is one of the design challenges in terms of power optimization and has been studied widely in the past.

Power minimization with V_{th} assignment and sizing has been presented in [2] and [3]. In [2] linear programming was used in combinational circuits and [3] used binary search to minimize power. Multiple V_{dd}, multiple V_{th}, gate sizing and force stacking methods were simultaneously applied in [4] using a genetic algorithm. Gate sizing and supply voltage optimization have been used in [5]. The effectiveness of using dual supply, dual threshold and device sizing has been shown in [6].

Our work tries to find best voltage and threshold voltages and device sizes with current technology for an automatic design tool which can be used when battery capacitance and power consumption is a limiting factor. Very Fast Simulated Reannealing has been used as a global optimization tool to achieve this goal.

The rest of the paper is organized as follows: Section 2 describes the problem and the condition for energy minimization and battery capacitance maximization. Section 3 describes application of very fast simulated reannealing. Section 4 includes the results on real-life models. Section 5 concludes the paper.

2 Preliminaries

The total power consumption of CMOS digital circuits may be represented by the following equation:

$$P = P_{dyn} + P_{leakage} + P_{sc} \tag{1}$$

The first term is the dynamic power consumption, which is due to charging and discharging of parasitic capacitance with the clock frequency. The second term is leakage power consumption mostly due to subthreshold leakage current. The last term is short circuit power consumption, which is due to short circuit current. Short circuit current flows for a short time during logic switching when both nmos and pmos transistors are ON and there is a direct path from supply voltage to ground. Short circuit power is 10% of the total power consumption and ignored in this work.

Currently, dynamic power consumption is the dominant power consumption and can be expressed by

$$P_{dyn} = \alpha C_L V_{dd}^2 f \tag{2}$$

Here, C_L is the load capacitance, α is the switching activity, V_{dd} is the supply voltage and f is the clock frequency. Load capacitance is a combination of parasitic capacitances and is a function of supply voltage. αC_L product refers to switched capacitance. To calculate the dynamic power of the whole chip, total switched capacitance in 1 second is multiplied by V_{dd}^2. Switching activity of each block varies, and is input dependent. Statistical methods were used widely to estimate switching activities.

Leakage power consumption is mostly due to the subthreshold leakage current, I_{leak}, which varies with processing technology.

$$I_{leak} = k\mu C_{ox}\frac{W}{L}V_T^2 e^{\frac{V_{gs}-V_{th}}{nV_T}}(1-e^{-\frac{V_{ds}}{V_T}}) \tag{3}$$

where μ is mobility, C_{ox} is the gate oxide capacitance per unit area, W is the channel width, L is the channel length, V_T is the thermal voltage, V_{th} is the threshold voltage, n is the subthreshold swing coefficient and k is a technogy dependent constant. Here V_{th} and temperature are exponentially and W is linearly dependent to leakage current.

Circuit delay is related to supply voltage and threshold voltage by the following formula:

$$T = k'C_L\frac{V_{dd}}{(V_{dd}-V_t)^{\alpha}} \tag{4}$$

Minimizing power does not necessarly maximize battery life. Peukert's equation [1] shows the nonlinear relationship between battery capacitance C and discharge current I,

$$C = T_d I^{\alpha} \tag{5}$$

where T_d is discharge time and α is called Peukert's constant which is typically in the [1.2,1.4] interval.

3 Method

3.1 Overview

In our optimization method, we try to minimize P/T_d, where P is total power and T_d is battery discharge time. We try to find optimum combination of V_{dd}, V_{th}, and gate size. The effectivity of parameters mostly depends on dynamic leakage current ratio and circuit topology. When extra slack is available, gate width can be reduced in order to reduce power. Reducing the width of each gate decreases the load capacitance of driving gates so further downsizing of the gate widths are possible.

Decreasing the suply voltage reduces dynamic power significantly since supply voltage is quadratically related to power. Leakage current also decreases linearly. Using high V_{th} devices decreases leakage power exponentially. Both methods come with the expense of increased delay. Dynamic to leakage power ratio of the whole chip determines which method is more effective.

Our algorithm tries to find best combination of V_{dd}, V_{th}, and gate size to minimize power and maximize battery discharge time. Since these computations require high complexity, Very Fast Simulating Reannealing algorithm has been used.

3.2 Very Fast Simulated Reannealing

Very Fast Simulated Reannealing (VFSR) is one of the better improved versions of the Simulated Annealing (SA) algorithm which is originated from the statistical mechanical model of atoms of metals being first heated and then cooled down slowly for a globally stable crystal form. An analogy can be formed between the evolution of the energy of each atom and the error function of a general optimization problem simulating the formation of the globally stable metal crystal during the SA process.

Simulated Annealing, before reaching the VFSR form, has gone through the stages of Boltzmann Annealing (BA) which uses a Boltzmann distribution to guide the heuristics and Fast Simulated Annealing (FSA) which uses a Cauchy Annealing.

In Very Fast Simulated Reannealing, transition rules are guided by the parameter y_i where a random number u_i is drawn from a uniform distribution U[0,1] which is then mapped as:

$$y_i = sgn(u_i - 1/2)T_i[(1+1/T_i)^{|2u_i-1|} - 1] \tag{6}$$

VFSR runs on an annealing schedule decreasing exponentially in time k,

$$T = T_0 e^{-ck^{1/D}} \tag{7}$$

This annealing schedule makes the algorithm faster than fast Cauchy annealing (FSA), where

$$T = \frac{T_0}{k} \tag{8}$$

and much faster than Boltzmann annealing, where

$$T = T_0/lnk \tag{9}$$

The introduction of re-annealing also permits adaptation to sensitivities of the parameters [7].

Although early versions of SA such as BA lacked from the curse of dimensionality, VFSR is a very fast tool which was shown to be much efficient than Genetic Algorithms (GA) for six nonconvex functions [8].

3.3 Cost Function

Our strategy performs three-fold power reduction through optimization of the total P/T_d of each noncritical path of the circuit as cost function. P/T_d is a function of Wn, Vth and T of all the elements of the noncritical paths. So the optimization is a search in a 3N dimensional space, where N is the number of circuit elements.

For the conquer part of a divide and conquer strategy, optimization by VFSR can be used in the conquer part, which is good for the optimization of circuits up to several hundred elements.

The calculation of the cost function involves the following procedure:

- Find the critical path on the circuit.
- Define the total delay.
- For each other path, do the following:
  - Define T_i, Wn_i and V_{thi} as free parameters of each element of the path
  - Where $\sum T_i$=mobility
  - Calculate V_{ddi} from the constraint equation
  - Impose V_{ddi} boundary conditions
- Calculate cumulative $cost = P/T_d$ for the whole circuit.

Table 1. % Results of the VFSR

Example	Power reduction	T_d increase	P/T_d improvement
c17	14.6 %	15.5 %	16.5 %
s27	18.2 %	19.1 %	20 %
b01	38.1 %	45.4 %	53.5 %
b02	33.7 %	34.2 %	34.8 %
Average	26.2 %	28.6 %	31.2 %

4 Results

We applied our method to some of the ISCAS'85 benchmark circuits. The average power reduction is 26.2%, while average battery discharge time increase is 28.6%. The objective function, P/T_d improved on average by 31.2% and simulation resulted in seconds in a P4 processor. We saw that gate sizing is the most effective way of optimizing the cost function, following by V_{dd} reduction and threshold voltage reduction. However when switching activity decreases, leakage power consumption dominates the total power consumption of the circuit so threshold voltage increment becomes more effective than supply voltage reduction in terms of power saving.

5 Conclusions

In this paper we addressed the problem of optimum supply and threshold voltage selection with device sizing by minimizing power consumption and maximizing battery charge capacitance using Very Fast Simulated Reannealing (VFSR). VFSR is very applicable to gate level optimization problems when there are multiple optimization variables available and there exist a timing dependency among the gates. Objective function is chosen as P/T_d since minimizing power not necessarily maximize battery charge time by itself.

We found that the most effective way of reducing power is gate sizing when dynamic power is comparable to leakage power. Supply voltage reduction is the second most effective design variable to minimize power and maximize battery discharge time. However this result will change when the leakage power dominates the total power consumption of the circuit. In that case the most effective way of reducing power is trading of delay with threshold voltages.

References

1. Linden, H.D.: Handbook of Batteries. 2^{nd} edition. McGraw-Hill, New York (1995)
2. Nguyen, D., et al.: Minimization of dynamic and static power through joint assignment of threshold voltages and sizing optimization. In: Proc. of the Int. Symp. on Low Power Electronics and Design. (2003)
3. Pant, P., Roy, R., Chatterj, A.: Dual-threshold voltage assignment with transistor sizing for low power CMOS circuits. IEEE Trans. on VLSI **9** (2001) 390–394

4. Hung, W., et al.: Total power optimization through simultaneously multiple-Vdd multiple-Vth assignment and device sizing with stack forcing. In: Proc. of the Int. Symp. on Low Power Electronics and Design. (2004)
5. Stojanovic, V., et al.: Energy-delay tradeoffs in combinational logic using gate sizing and supply voltage optimization. In: Proc. European Solid-State Circuits Conf. (2002)
6. Augsburger, S., Nikolij, B.: Reducing power with dual supply, dual threshold, and transistor sizing. In: Proc of Int. Conf. on Comp. Design. (2002) 316–321
7. Ingber, L.: Very fast simulated reannealing. Mathl. Comput. Modeling **12** (1989) 967–993
8. Ingber, L., Rosen, B.: Genetic algorithms and simulated reannealing. Mathl. Comput. Modeling **16** (1992) 87–100

Compressed Swapping for NAND Flash Memory Based Embedded Systems

Sangduck Park, Hyunjin Lim, Hoseok Chang, and Wonyong Sung

School of Electrical Engineering,
Seoul National University,
Gwanak-gu, Seoul 151-742 Korea
{parksd, hjlim, chs, wysung}@dsp.snu.ac.kr

Abstract. A swapping algorithm for NAND flash memory based embedded systems is developed by combining data compression and an improved page update method. The developed method allows efficient execution of a memory demanding or multiple applications without requiring a large size of main memory. It also helps enhancing the stability of a NAND flash file system by reducing the number of writes. The update algorithm is based on the CFLRU (Clean First LRU) method and employs some additional features such as selective compression and delayed swapping. The WKdm compression algorithm is used for software based compression while the LZO is used for hardware based implementation. The proposed method is implemented on an ARM9 CPU based Linux system and the performances in the execution of MPEG2 decoder, encoder, and gcc programs are measured and interpreted.

1 Introduction

Embedded systems nowadays have become very powerful to support demanding applications. This inevitably brought an increase in main memory capacity. The flash memory device is a critical component in building these systems because of its nonvolatility, shock-resistant, and power-economic nature. A typical embedded multimedia system such as a high-end cellular phone usually contains DRAM, NOR flash and NAND flash memory devices where DRAM is used as a working memory, NOR flash as a code storage and NAND for non-volatile data storage. Recently, it has been attempted to eliminate the costly NOR flash memory. In this case, OS and application programs need to be transferred from NAND flash to the main memory during the boot time and such process is called ”shadowing.” The shadowing offers the best performance at runtime but it needs a longer loading time because of the copy overhead. It also requires a larger DRAM because the DRAM should provide the space for both working memory and the code.

An alternative to the shadowing is “demand paging” where pages of code or data are copied from the secondary storage to the main memory only when they are needed [1]. Thus it demands a less DRAM size and a smaller loading time. However it needs a page swapping algorithm and that may result in a poor system performance because of the overhead of swapping. Another concern with the NAND flash based swapping is

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 314–323, 2005.

the possible degradation of the system because the number of writes is limited to about 100,000 times. In addition, it needs to be considered that the write cost for evicting a page is much higher than the read cost in NAND flash memory. Because of these reasons, the study on NAND flash memory based swapping has not been explored much.

The basic idea of this work is reducing the number of page writes using compressed swap as well as optimization of the update algorithm. In this study, a swapping method that can not only reduce the overhead but also the number of writes to NAND flash memory is developed.

The rest of this paper is organized as follows. Next section describes the characteristics of NAND flash memory and previous related works. In Section 3, the developed compressed swap algorithm is explained. In section 4, experimental results on ARM9 CPU based Linux environment with some practical applications are shown. Finally, concluding remarks are shown in Section 5.

2 Background

2.1 Characteristics of the Storage Devices

A most widely used non-volatile memory device would be the hard disk which can provide a large capacity at a cheap cost-per-byte. But hard disks consume much power and are less robust to physical stresses. NAND flash memory is a non-volatile, high density semiconductor device and is usually used for small hand-held devices, such as MP3 players and digital cameras [2]. Although it provides the highest density compared to NOR flash and DRAM, its contents are not random accessible. A flash memory device contains a fixed number of blocks and a block consists of usually 16 to 64 pages. Each page normally consists of 512 bytes of main data and 16 bytes of spare data although the page size has an increasing tendency as the density of the flash memory goes up. For example, a typical 64MByte NAND flash contains 4K blocks, each with 32 pages and a page contains 512 bytes for data.

Table 1. Characteristics of memory [3][4]

Device	current(mA)		Access time(4kB)		
	Idle	Active	Read	Write	Erase
NOR	0.03	32	23us	28ms	1.2sec
NAND	0.01	10	291us	1.8ms	2ms
SDRAM (32MB)	0.50	85	184us	184us	N/A
SDRAM (64MB)	1.00	120	184us	184us	N/A
Hard disk (20GB)	23	420	15ms	15ms	N/A

In order to read a page, commands are given to the NAND flash memory through I/O pins because read and write operations are conducted on a page basis. The write operation of NAND flash memory is a little bit complicated since it is only allowed to erased pages. Note that the erase operation is conducted on a block basis but writes are allowed on a page basis. Another restriction is that the number of updating a block is limited to about 100,000 times in a typical single cell NAND flash memory. Because of these reasons, NAND type flash memory needs a file system that supports a wear level control such as a journaling file system.

Table 1 shows the characteristics of various types of memory found in embedded systems. In case of NAND flash, the write operation takes longer, about 6 times the latency, compared to the read although it has advantages in terms of power consumption and storage capacity. Mobile SDRAM shows a fast read/write performance but requires higher power consumption over the other memories. It is notable that an SDRAM device with a larger capacity requires a higher power not only in the idle time but also in the active mode. Thus, the demand paging based system implementation is very attractive for reducing the power consumption without limiting the possibility of executing a large application [5].

2.2 Compressed Swap

Traditional computer systems mainly use hard disks as a secondary storage. A typical disk access latency is around tens of milliseconds as shown in Table 1, which is much larger than the access time of other types of memory. Hence there had been many researches for reducing the number of hard disk accesses by compressing pages being swapped. Compressed swap systems can be implemented as either software [6, 7, 8, 9] or hardware [10, 11, 12, 13, 14, 15] . A software based approach is shown in Fig. 1.

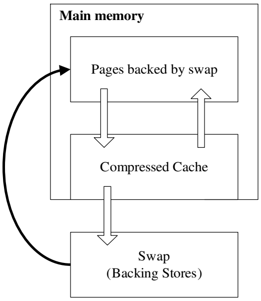


Fig. 1. Software based compressed swap system

2.3 CFLRU (Clean First LRU) Algorithm

Traditional operating systems mainly use the LRU (Least Recently Used) or pseudo-LRU page replacement whose primary goal is to minimize the hard disk accesses. However as shown in Table 1, the write and read costs are not the same for the NAND flash memory devices. Thus it is important to reduce the number of writes even if it incurrs more read operations. In the CFLRU algorithm, dirty pages are kept as long as possible so that the number of swap outs can be minimized [16]. Dirty pages mean memory blocks that have been modified, thus they need to be newly compressed and stored to the file system as being swapped out. As the page fault ratio increases, only the clean pages within the predetermined window size become candidates for a victim of the CFLRU. If it does not find any clean pages within the window, it turns into a regular LRU algorithm and swaps out dirty pages.

3 Compressed Swap Systems for NAND Flash

3.1 Compression Algorithms for Data

We employed two well known algorithms for file compression, WKdm and LZO. The WKdm developed by Paul Wilson and Scott Kaplan, shows a fast compression performance [9]. The LZO is an improved implementation of the well known Lempel-Ziv algorithm.

Table 2. Performance of the compression algorithms

	WKdm	LZO
Compress time (throughput)	248μs (16.5MB/s)	904μs (4.5MB/s)
Decompress time (throughput)	216μs (18.0MB/s)	233μs (17.6MB/s)
Compression ratio		
mpeg4decode	71%	51%
mpeg4encode	39%	21%

Table 2 shows the performance of the two compression algorithms for MPEG4 decoder and MPEG4 encoder. Note that the compression ratio (CR) is defined as compressed data size over the uncompressed data. Thus a smaller compression ratio means an efficient compression. Table 2 shows that LZO yields a better compression, however requires approximately three times more execution time. Considering the write time of 1.8ms for a 4KB block in a typical NAND flash memory device, the overhead of 904μs for the software based implementation is too high. Thus the WKdm algorithm is used for the software based compression and the LZO in the hardware based one. Note that the compression and decompression speed becomes more important in the NAND flash memory based systems because the access time of the NAND flash memory is much shorter than that of the hard disks.

3.2 Selective Compressed Swapping

While most hard disk based systems conduct read and write operations in the cluster size of 4KB, the page size for NAND flash memory devices is typically 512B. Therefore as the compressed page size decreases, it is possible to reduce the number of page writes [17]. For example, if the compression ratio is 50% which means 2KB of the compressed size, only four pages are needed for writing. If the compression ratio is too high which

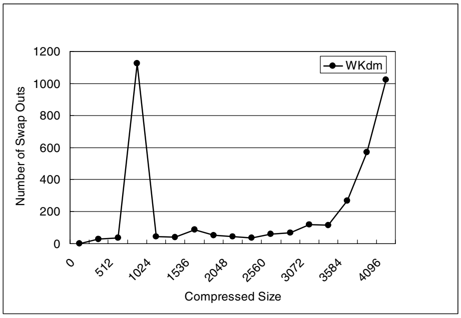


Fig. 2. Mpeg2decode distribution of compressed page size

means poor compression, it is not much beneficial to store the compressed pages since decompression also demands CPU power while not reducing the IO operations much. Therefore we store uncompressed pages when the compression does not yield much data size reduction.

As expected, the compressed size for data is very much different in each page as exemplified in Fig. 2.

3.3 Update Algorithm for Reducing the Number of Writes

Since NAND flash memory has the limitation in the number of erases, a modification of swap algorithm is needed to reduce the number of writes. The developed algorithm is similar to that of CFLRU since it can reduce the number of writes by discarding the clean pages first. However the original CFLRU algorithm can lead to a worse performance when dirty pages can be compressed very efficiently. In this case, it would be advantageous to swap out highly compressible dirty pages.

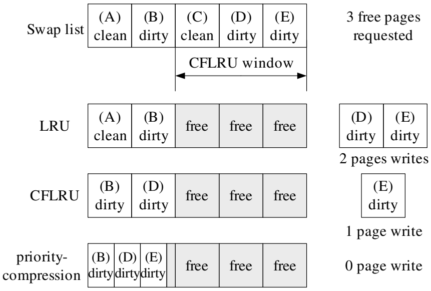


Fig. 3. Comparison of swap outs according to update algorithms

In order to resolve this problem, swap out is conducted only when the swap out priority is at its highest level. In this case, the compression cache acts as a sacrifice buffer and leads to less number of writes. In the Linux swap system, the priority level is increased when a required amount of free pages are not obtained while the swap daemon executes [18, 19, 20]. On the other hand, if it has enough free pages, the priority decreases.

The original CFLRU algorithm may swap out pages even when free pages can be reserved by compression as illustrated in Fig. 3. In a proposed system, CFLRU algorithm is applied after compressing the pages to reduce the number of page writes to the NAND flash based file system.

In the case of the compressed swap system, the delayed write is applied which can further reduce a significant amount of writes. This is due to the characteristic of the application programs which repeatedly create and discard many temporary buffers, where swap out operations are delayed in many cases until the temporary buffers are discarded.

4 Experimental Results

4.1 Experimental Environment

The experiment was conducted with Samsung's S3C2410 CPU based system containing an ARM920T core with 200MHz clock, 64MB SDRAM and a NAND flash memory based file system. Linux kernel of version 2.4.18 was used . The experiment with the limited main memory size is also conducted by utilizing the memory restriction feature. The size of NAND type flash memory used in the experiment was 32MB with the page size of 512B. The root file system for Linux, YAFFS, was stored in the NAND type flash region. For the swap device, 8MB swap file was used and was configured in the NAND type flash region.

Mpeg2decode, mpeg2encode and gcc are used as application programs for the experiment. Note that mpeg2encode is a demanding application while gcc contains mostly clean pages. The number of writes and the time taken for writing to NAND type flash was measured. The number of writes is also important because of the restrictions in the number of erases in NAND flash memory devices.

As for the number of levels for swapping priority, the default value of 6 was used although the optimal value may be different according to application programs.

4.2 Threshold Determination for Selective Compressed Swapping

The most important variable is α that shows the threshold value for the selective compressed swapping. Note that when the compressed page size is larger than α, the system swaps out the uncompressed page in order to reduce the decompression overhead at the read time. Figure 4 shows the execution time according to α for mpeg2decode. In this case, the optimum value for α is between 2,560 and 3,584 because the compressed swapping incurs the decompression overhead as well. Since the variation of the performance is small, the threshold value is set to 3,584 to utilize the compressed swapping at its maximum.

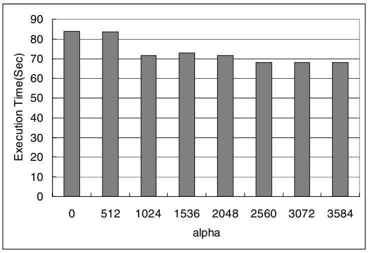
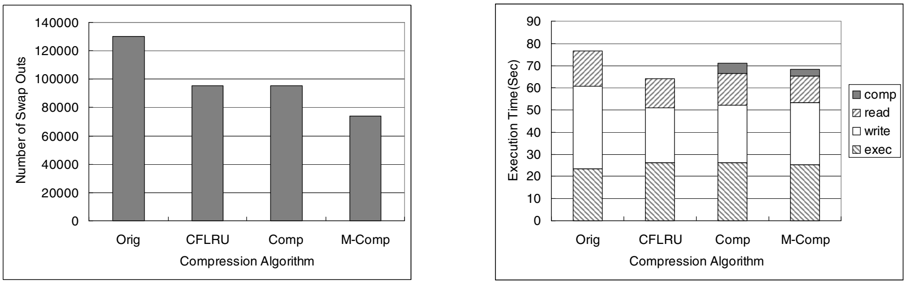


Fig. 4. The execution time according to α

4.3 Performance Evaluation

Mpeg2decode decodes an MPEG2 file and outputs the result as a file. The system memory size used is 4MB. The original one (Orig) which is based on LRU without compression, the CFLRU (CFLRU) without compression, the CFLRU with software compression (Comp) and the modified CFLRU with software based compression (M-Comp) are compared. The software based compression uses the WKdm algorithm.



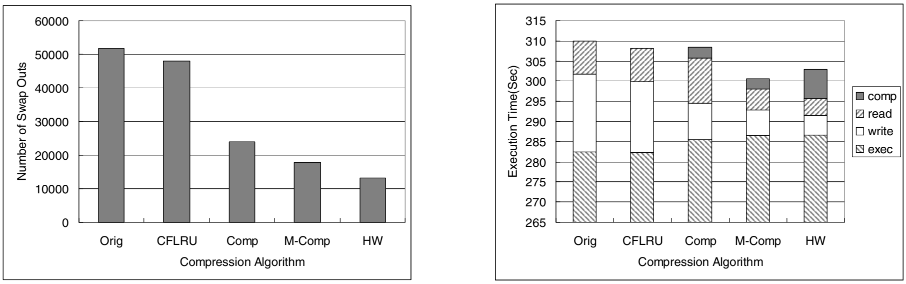
(a) Number of swap outs (b) Execution time

Fig. 5. Experimental results of mpeg2decode

Figure 5(a) shows the number of write operations in each swapping algorithm. The modified update algorithm with software compression results in about 43% and 23% of reduction in the number of writes when compared to the original uncompressed swap system and the CFLRU based one, respectively. Figure 5(b) illustrates the execution time according to the employed swapping methods. It shows that the execution time of the software based compression with modified swapping is a little, 6%, longer than that of the CFLRU based one because of the overhead in the compression.

Comparing the compression only and CFLRU algorithms, the number of writes are almost the same however the execution time of the compression based one is longer because of the overhead in the software based compression.

Figure 6 shows the case for MPEG2 encoder. The hardware based compression(HW) that employs the LZO is also compared. Note that this application shows quite low com-



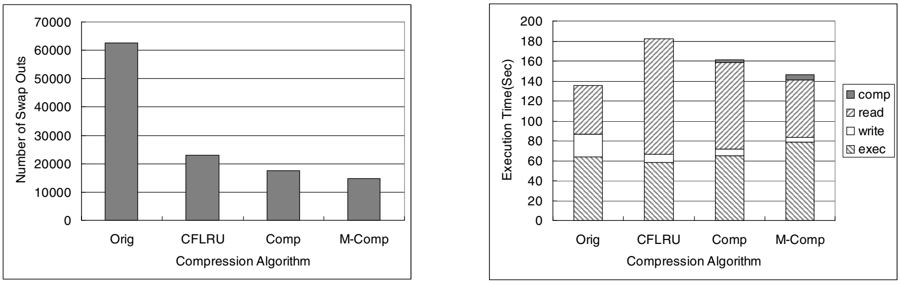
(a) Number of swap outs

(b) Execution time

Fig. 6. Experimental results of mpeg2encode

pression ratio which means a good compression. As a result, it is possible not only to reduce the number of page writes but also the execution time as well even with the software compression. As shown in this figure, the number of page writes and the execution time are reduced to approximately 62% and 3% respectively when compared to the CFLRU algorithm. The performance gain obtained by using CFLRU is less compared to the previous experiment because the application program has a long execution time and a large spatial locality. However there is a great improvement in number of swap outs.

The performance of gcc in spec2000 benchmark programs is also evaluated and the results are illustrated in Fig. 7.

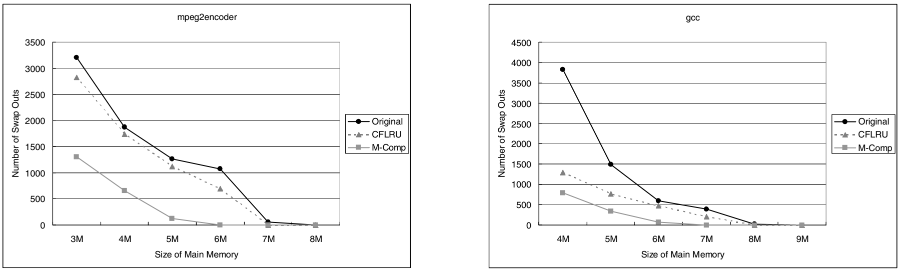


(a) Number of swap outs

(b) Execution time

Fig. 7. Experimental results of gcc, main memory 4MB

The result shows that when CFRLU is applied, although the number of writes has been reduced, the execution time has increased significantly compared to other experiments. This is because gcc makes a lot of reference to data and in the case of CFLRU, clean pages are swapped out first. Thus the average duration residing in the main memory is reduced and the number of reads for referring data increases rapidly. In the proposed algorithm, the efficiency in memory usage is increased therefore it is possible to reduce the number of writes and minimize the overhead increase due to read. When compared to CFLRU, the number of writes in the compression based algorithms is much less because the compression performance of this application is very good.



(a) mpeg2encoder

(b) gcc

Fig. 8. Changes in the number of swaps according to main memory size

The number of swaps according to different main memory size is also observed and the results shown in Fig. 8.

5 Concluding Remarks

In this paper, a swapping method combining the file compression and enhanced swap scheduling is introduced for NAND flash based virtual memory systems. The results show that it can reduce the number of writes significantly, 50% $\sim$ 70%, without affecting the system performance much. Since the total number of writes allowed in NAND flash is limited, the proposed method enhances the stability of a system. Reduced DRAM size helps for lowering the system implementation cost as well as power consumption. This research also suggests the direction for improved NAND flash architecture design that is optimized for demand paging applications.

References

1. Park, C., Seo, J., Bae, S., Kim, H., Kim, S., Kim, B.: A low-cost memory architecture with nand xip for mobile embedded systems. In: CODES+ISSS. (2003) 138–143
2. Kim, J., Kim, J.M., Noh, S.H., Min, S.L., Cho, Y.: A space-efficient flash translation layer for compactflash systems. IEEE Trans. Consumer Electronics **48** (2002) 366–375
3. Samsung Electronics: NAND flash memory&SmartMedia data book. (2004)

4. Samsung Electronics: Mobile SDRAM (K4S561633F, K4S511633F) data sheets. (2004)
5. Marsh, B., Douglis, F., Krishnan, P.: Flash memory file caching for mobile computers. In: HICSS (1). (1994) 451–461
6. de Castro, R.S., do Lago, A.P., Silva, D.D.: Adaptive compressed caching: Design and implementation. In: SBAC-PAD. (2003) 10–18
7. Cervera, R., Cortes, T., Becerra, Y.: Improving application performance through swap compression. In: USENIX Annual Technical Conference, FREENIX Track. (1999) 207–218
8. Roy, S., Kumar, R., Prvulovic, M.: Improving system performance with compressed memory. In: IPDPS. (2001) 66
9. Wilson, P.R., Kaplan, S.F., Smaragdakis, Y.: The case for compressed caching in virtual memory systems. In: USENIX Annual Technical Conference, General Track. (1999) 101–116
10. Abali, B., Banikazemi, M., Shen, X., Franke, H., Poff, D.E., Smith, T.B.: Hardware compressed main memory: Operating system support and performance evaluation. IEEE Trans. Computers **50** (2001) 1219–1233
11. Nunez, J.L., Feregrino, C., Jones, S., Bateman, S.: X-matchpro: A proasic-based 200 mbytes/s full-duplex lossless data compressor. In: FPL. (2001) 613–617
12. Bunton, S., Borriello, G.: Practical dictionary management for hardware data compression. Commun. ACM **35** (1992) 95–104
13. Kjelsø, M., Gooch, M., Jones, S.: Design and performance of a main memory hardware data compressor. In: EUROMICRO. (1996) 423–430
14. Kjelsø, M., Gooch, M., Jones, S.: Performance evaluation of computer architectures with main memory data compression. Systems Architecture **45** (1999) 571–590
15. Jones, S.: 100mbit/s adaptive data compressor design using selectively shiftable content-addressable memory. In: Proceedings of IEE (part G). (1992) 498–502
16. Park, C., Kang, J.U., Park, S.Y., Kim, J.S.: Energy-aware demand paging on nand flash-based embedded storages. In: ISLPED. (2004) 338–343
17. Lee, J.S., Hong, W.K., Kim, S.D.: Design and evaluation of a selective compressed memory system. In: ICCD. (1999) 184–191
18. Rusling, D.A.: The Linux Kernel. O'Reilly (1999)
19. Bovet, D., Cesati, M.: Understanding the Linux Kernel. O'Reilly (2002)
20. van Riel, R.: Page replacement in linux 2.4 memory management. In: USENIX Annual Technical Conference, FREENIX Track. (2001) 165–172

A Radix-8 Multiplier Design and Its Extension for Efficient Implementation of Imaging Algorithms

David Guevorkian[1], Petri Liuha[1], Aki Launiainen[1], Konsta Punkka[2], and Ville Lappalainen[3]

[1] Nokia Research Center, Visiokatu 1, FIN-33721 Tampere, Finland
David.Guevorkian@Nokia.com
[2] Tampere University of Technology, P.O.Box 553, FIN-33101 Tampere, Finland
[3] Nokia Multimedia Business Unit, Tampere, Finland

Abstract. In our previous work, general principles to develop efficient architectures for matrix-vector arithmetics and video/image processing were proposed based on high-radix (4,8, or 16) multiplier extensions. In this work, we propose a radix-8 multiplier design and its extension to *Multifunctional Architecture for Video and Image Processing* (MAVIP). MAVIP may operate either as a programmable unit with DSP-specific operations such as multiplication, multiply-accumulate, parallel addition or as one or another HWA such as matrix-vector multiplier, FIR filter, or sum-of-absolute-difference accelerator. Simulations indicate that being a small device, MAVIP has competitive performance in video coding.

1 Introduction

Research for efficient multiplier structures is as old as digital computers and still is a hot topic attracting many researchers due to its importance to the efficiency of computations and due to many possible ways of implementing multiplication operations (see [1]-[2]). Though the relative efficiency of different types of multipliers changes with the technology development, probably the most known multiplier type remains to be the Booth recoded radix-4 multiplier structure (see [2], [3]). In [1], [4], [5], [6], higher-radix (radix-8, radix-16, etc.) multipliers for nonnegative integers based on generalized Booth recoding were proposed. Mixed radix multipliers [7], [8], [9] (radix-4 and 8) multipliers were also suggested.

While Booth recoded radix-4 multipliers achieve lower latency implementations of multiplications, the use of higher radices may significantly reduce the implementation cost and power consumption [1]. However, the higher radix methods did not gain much popularity. The main reason for this is that these higher-radix multipliers involve a number of addition/subtractions for finding a list of potential partial products at the first step. This, in general, means larger area and slower implementation of standard multiplication operations. This also means that in a pipelined higher-radix multiplier design, the first pipeline stage will be significantly more complex compared to the consecutive ones (meaning a poor balancing) as well as compared to the case of conventional pipelined radix-4 Booth recoded multiplier.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 324–333, 2005.

In our previous work [10], [11], we have proposed a methodology that turns these drawbacks to advantages in some cases such as matrix-vector products (in particular, discrete orthogonal (Cosine Fourier) transforms), scalar to matrix/vector products, FIR filtering, *etc*. Three main principles of this methodology can be summarized as follows. First, in order to decrease the implementation time in those cases where one multiplier is to be multiplied with a numerous multiplicands, we omit the first step in most of the multiplication operations by reusing the potential partial products that are once computed and are stored. Second, to reduce the effect of grown area we reuse adder/subtractors involved in the first pipeline stage of a high-radix multiplier to implement other operations thus eliminating the need of separate adders/subtractors that are usually involved in the system. Third, we propose a principle of balancing between stages of pipelined devices, which may efficiently be applied to higher-radix multiplier structures to achieve perfect balancing.

In this work, we apply the principles of [10], [11] to a novel radix-8 non-recoded multiplier structure to derive an example realization of so called Multifunctional Architecture for Video and Image Processing (MAVIP). This realization of MAVIP has been modelled with VHDL showing competitive performance in video coding. In Section 2 the design of the radix-8 multiplier is presented. Section 3 presents MAVIP realization as extensions to the radix-8 multiplier. Section 4 presents performance analysis. Conclusions are given in Section 5.

2 The Radix-8 Multiplier Structure

Basically, multiplication is a process of finding partial products and adding those together. Multiplication algorithms and multiplier structures differ in the way of partial product generation and summation. The earliest algorithms were based on radix-2 method where the product $y = a \cdot x$ is obtained by adding n shifted instances $a_i x 2^i$, $i = 0,...,n-1$, of the multiplicand x masked by the bits a_i of the multiplier $a = a_{n-1}$ a_{n-2} $\ldots$ a_1 a_0. Additions are performed iteratively or in parallel using different adder trees (see [2], [3]). Later on, recoded multipliers were proposed where the multiplier a is Booth recoded to an alternative signed digital representation in a higher than 2 radix (see [2]-[9]). The most popular case is the radix-4 Booth-recoded multiplier where the product is obtained as:

$$y = \sum_{r=0}^{n_{radix-4}-1} (A_r x)\, 2^{2r} = \sum_{r=0}^{n/2-1} ([-2a_{2r+1} + a_{2r} + a_{2r-1}]x)\, 2^{2r}, \quad a_{-1} = 0$$

where the value of $A_r \in \{-2,-1,0,1,2\}$, $r = 0,1,...,n/2-1$, is chosen according to three consecutive bits $a_{2r+1}, a_{2r}, a_{2r-1}$ of the two's complement representation of the multiplier a. PP's that are non-negative multiples of x are readily available ($2x$ is formed by a hardwired shift). PP's that are negative multiples of x are formed by inverting the bits of the corresponding positive multiples of x and then by adding 1.Addition of PP's and these 1's is usually implemented in an adder tree that suppresses the $n/2$ input PP rows as well as these $n/2$ 1-bit signals into the two output rows (Sum S and Carry C terms). A fast final adder then completes the multiplication by adding the S and C terms.

The main advantage of the radix-4 Booth recoded multipliers is the reduced number $n/2$ of PP's compared to n PP's to be added in radix-2 multipliers. This leads to significant advantages in speed performance, area, and power consumption.

The number of PP's may be further reduced in higher-radix multipliers. In [1], [4]-[9], higher radix multipliers using generalized multibit recoding were considered. Similarly, non-recoded higher-radix multipliers may be developed. Below we present a radix-8 non-recoded multiplier design based on the following proposition.

Let $a = a_{n-1}a_{n-2}...a_1a_0$ and $x = x_{m-1}x_{m-2}...x_1x_0$ be the two's complement representations of the n-bit multiplier a and the m-bit multiplicand x, respectively. Let also $n = 3n'$ (n' is an integer[1]).

Proposition 1. *The product $y = a \cdot x$ can be obtained as:*

$$y = \sum_{r=0}^{n'-1} Y_r 2^{3r} + 2^k \tilde{X}$$

where $Y_r = < A_r \cdot x >_{(m+2)}$, $A_r = a_{3r+2}a_{3r+1}a_{3r} = 4a_{3r+2} + 2a_{3r+1} + a_{3r}$, $r = 0, ..., n'-1$, are PP values formed from the least $m+2$ significant bits of the two's complement representation of the number $A_r \cdot x$, $k = \min\{n, m+2\}$, and $\tilde{X}$ is the sign correction term given by

$$\tilde{X} = \begin{cases} x_{m-1}C(n)2^{m+2-k} & \text{if } a_{n-1} = 0 \\ x_{m-1}C(n)2^{m+2-k} - x2^{n-k} & \text{if } a_{n-1} = 1 \end{cases},$$

where $C(n) = -\frac{2^n-1}{7} = 1\ \underbrace{011\ 011\ ...\ 011}_{n'-1 \text{ times}}\ 1$. In the case $A_r = 0$, the value of the corresponding PP should be presented as $Y_{r=} \underbrace{x_{m-1}...x_{m-1}}_{m+2 \text{ times}}$ ("signed zero").

Proof. The radix-8 representation of the multiplier a is $a = \sum_{r=0}^{n'-1} A_r 2^{3r} - a_{n-1}2^n$. The product is obtained as $y = ax = \sum_{r=0}^{n'-1} (A_r x)\, 2^{3r} - (a_{n-1}x)\, 2^n$. In a direct computation, the terms $(A_r x)2^{3r}$ must have been sign extended to $n+m$ bits. However, since $A_r \geq 0$ (which is not the case in Booth recoding), one can present the value of the $(m+3)$-bit number $(A_r x)$ having in its $(m+2)$-nd position the same sign bit x_{m-1} as the multiplicand x as $(A_r x) = \langle A_r x \rangle_{m+2} - x_{m-1}2^{m+2} = Y_r - x_{m-1}2^{m+2}$ $(A_r x = x_{m-1}...x_{m-1}$ for $A_r = 0)$. Therefore,

$$y = \sum_{r=0}^{n'-1} Y_r 2^{3r} - x_{m-1}2^{m+2}\frac{2^n - 1}{7} - (a_{n-1}x)2^n$$

The proof is complete after combining the two rightmost terms.

[1] Multiplier may be sign extended if needed.

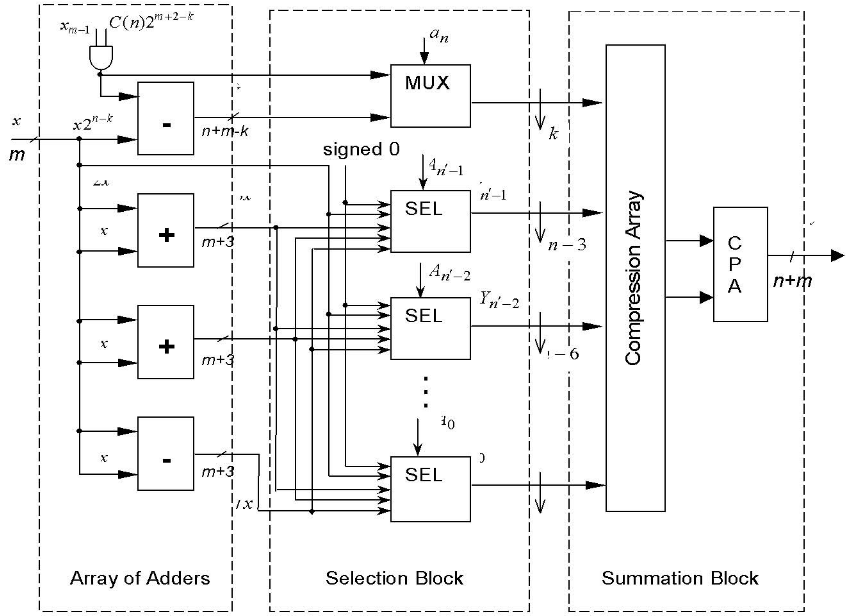


Fig. 1. The proposed radix-8 multiplier structure

Fig. 1 presents the proposed multiplier structure that implements the following algorithm, based on Proposition 1.

Algorithm 1.

Step1. Find all potential partial products $3x = 2x + x$, $5x = 4x + x$, and $7x = 8x - x$ of all 3-bit positive numbers with the multiplicand x. Also find $\hat{X} = x_{m-1}C(n)2^{m+2-k} - x2^{n-k}$. Note that every potential PP is the result of one addition or subtraction with one operand being x, and the other being x shifted to the left. This step is implemented in an array of four parallel adders/subtractors of the multiplier in Fig. 1 in one operating step.

Step 2. According to every group of three bits $A_r = a_{3r+2}a_{3r+1}a_{3r}$, $r = 0,...,n' - 1$, of the multiplier select n' partial products $Y_r = < A_r \cdot x >_{(m+2)}$ from the list of all possible partial product values 0, x, $2x, 3x$ $4x$, $5x$, $6x$, and $7x$ obtained at the first step. Also, if $a_{n-1} = 0$, then set the value of the sign correction term $\tilde{X} = x_{m-1}C(n)2^{m+2-k}$, otherwise (if $a_{n-1} = 1$), set $\tilde{X} = \widehat{X}$ where $\widehat{X}$ was obtained at the first step. This step is implemented in the Selection Block (SB) of the multiplier in Fig. 1. Different realizations of SB are possible. In the example of Fig. 1, SB consists of a m-bit 2 : 1 multiplexer and n' SEL units. The multiplexer has $\widehat{X}$ at its first input and $x_{m-1}C(n)2^{m+2-k}$ at its second input, and is controlled by a_{n-1} so that it produces $\tilde{X} = x_{m-1}C(n)2^{m+2-k} - a_{n-1}x2^{n-k}$. Every SEL unit has the values of 0, x, $3x$, $5x$, and $7x$ at its inputs and is controlled by corresponding three bits $A_r = a_{3r+2}a_{3r+1}a_{3r}$ of the multiplier a.

Step 3. Find the product y by summing up the n' selected partial products Y_r, $r = 0,...,n' - 1$, and the value of $\tilde{X}$ preliminary shifting Y_r by $3r$ and the value of $\tilde{X}$ by k

positions to the left. Step 3 is implemented in the summation block of the multiplier in Fig. 1, which is composed of a compression array (CA) followed by an adder (later on referred to as final adder). CA reduces the partial product rows into two rows that are then added in the final adder. Clearly, the presented multiplier can easily be extended to multiply-accumulate (MAC) unit if the result of the Step 3 is accumulated. For this, feedbacks from the outputs of CA to its inputs may be introduced. This way, partial multiplications excluding the final additions will be executed in a series of MAC operations.

Note that there are total of $n' + 1 = \lceil n/3 \rceil + 1$ PPs to be added in the radix-8 multiplication method (while in the radix-4 Booth multiplication, this number is $\lceil n/2 \rceil$ or $\lceil n/2 \rceil + 1$ depending on how the sign extensions are handled). Moreover, some of the $n' + 1$ rows may be merged. For example, the two rows 000101 and 011000 can be merged into the following one: 011101. It can easily be shown that, after merging the total number of PP rows is $\min\{\lceil m/3 \rceil, \lceil n/3 \rceil\} + 1$.

Also note that no sign extensions are needed in implementing the summation since all the PP's Y_r are nonnegative and $2^k\tilde{X}$ is an $(m+n)$-bit number. As the numbers of rows to be added are reduced when using the radix-8 method instead of radix-4 method, the numbers of levels within the Compression Arrays and, therefore its delay and size are significantly reduced. For example, in the case of 13-bit multiplicand $(m = 13)$ and 16-bit multiplier $(n = 16)$ the numbers of levels of the CA in the radix-8 multiplier and in the radix-8 MAC unit are reduced from 4 to 3 and from 5 to 4 when comparing to the state-of-the-art radix-4 Booth multipliers and MAC units, respectively. The number of full adders (FAs) and the number of half adders (HAs) are reduced from 72 and 14 to 36 and 13.

3 An Extension of the Radix-8 Multiplier, MAVIP

In this section, we apply the methodology of [10], [11] to the proposed multiplier design in order to derive an efficient architecture operations that are frequently used in image and video processing algorithms. The resulting architecture called Multifunctional Architecture for Video and Image Processing (MAVIP) is depicted in Fig. 2. This realization is based on the (13x16)-bit multiplier $(m = 13, n = 16)$ where the following extensions/modifications are applied.

Firstly, the array of four adders/subtractors of the multiplier is transformed to so-called "reconfigurable" array of eight 8-bit Array of Adders or Difference-Sign units (AA/DSA) which is controlled by two signals c_1 and c_2. A possible realization of a unit within the AA/DSA block is shown on Fig. 3. Depending on the control signals c_1, c_2 four of such units involved in the AA/DSA block may implement either eight 8-bit additions/subtractions, or four 16-bit addition/subtraction, or eight difference-sign operations, or the two subtractions and the two additions of the first step in (13x16)-bit radix-8 multiplication algorithm. The difference-sign operation is needed to compute the sum-of-absolute-differences (SAD), which is the base of motion estimation, the most complex part of video encoders (see [12]). Given two arrays $a_{i,j}$ and $b_{i,j}$, $i = 0,...,N$, SAD is defined as $SAD(a,b) = \sum \left|a_{i,j} - b_{i,j}\right|$. It can be shown that $SAD(a,b) = \sum \left(\tilde{d}_{i,j} + s_{i,j}\right)$ where $s_{i,j}$ is the sign bit of the difference $a_{i,j} - b_{i,j}$, and $\tilde{d}_{i,j} = d_{i,j}^{n-1} \oplus$

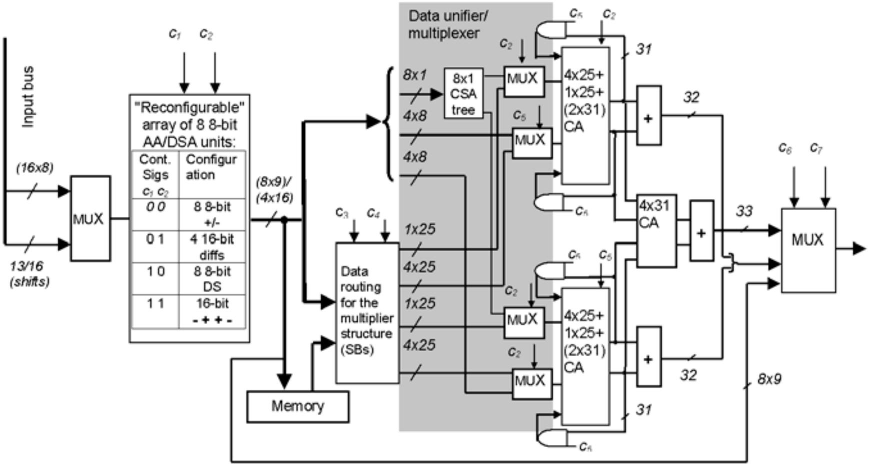


Fig. 2. A MAVIP realization

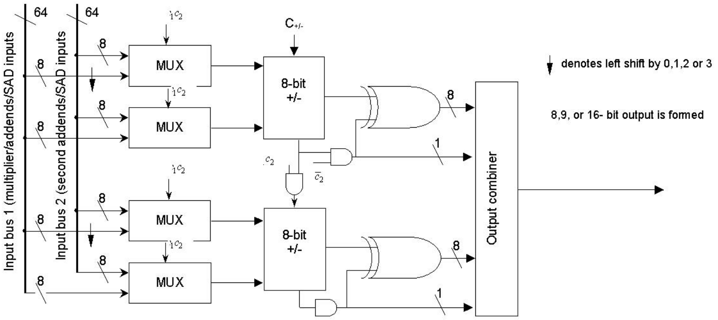

64
64
Input bus 1 (multiplier/addends/SAD inputs
Input bus 2 (second addends/SAD inputs
8
MUX
c2
C+/-
8-bit +/-
1
Output combiner
denotes left shift by 0,1,2 or 3
8,9, or 16- bit output is formed

Fig. 3. An example realization of a unit within AA/DSA

$s_{i,j}d_{i,j}^{n-2} \oplus s_{i,j}...d_{i,j}^{0} \oplus s_{i,j}$ is formed by XORing every significant (non-sign) bit of $a_{i,j} - b_{i,j}$ with $s_{i,j}$. The difference-sign operation for a pair of n-bit operands outputs the corresponding n-bit $\tilde{d}$ and 1-bit s values (all these values will later be added together in the summation block to form the SAD).

Second extension consists of introducing a register memory for storing the lists of potential PP's obtained in multiplication operations. The aim of this extension is to reuse these lists whenever one multiplier is to be multiplied with many multiplicands. One such example is matrix-vector product where every component of the input vector is multiplied with corresponding column of the input matrix. Another example is FIR filtering where every filter coefficient is multiplied with every sample of the input signal. Reusing the list of PP's allows excluding the most costly first step from multiplication process thus saving the execution time and the energy consumption.

Next extension consists of modifying and duplicating the compression array (CA) as well as introducing a data unifier/multiplexer block. The idea of modification is to derive a unified CA that supports both compression of five PP rows for multiplication and compression of eight 8-bit rows and eight 1-bit rows for SAD computation. The input to the data unifier/multiplexer may be either forwarded directly from the AA/DSA block or taken from the memory depending on control signals c_3 and c_4. Modification also consists of introducing "closeable" feedback loops between outputs and inputs of CAs, which are either open or close depending on control signal c_5. When the loop is open, MAC operations are implemented otherwise simple multiplications are implemented. Duplicating of CA's of the multiplier structure is needed because there are about twice more rows to be compressed in the case of SAD compared to multiplication. At the same time, it allows of implementing two multiplication or MAC operations with a shared operand (multiplier) in one cycle. The outputs of the two CA's may either be separately outputted to two separate final adders or combined together using the third CA (4x31 CA) and final adder.

The last modification consists of introducing the output selector-multiplexer, which allows of reusing the AA/DSA block of the first stage as independent array of adders/subtractors. Reusing AA/DSA block may, in many cases, eliminate the need in separate adders within the system where the MAVIP is supposed be used.

MAVIP may be configured to one or another hardware accelerator (HWA) by halting or activating its blocks (AA/DSA, block of CA's and the final adders), and by setting the control signals. For example, it may be configured to a HWA for FIR filtering. In this case, lists of potential PP's of the filter coefficients are pre-computed and loaded to the MAVIP's internal memory block. Then the AA/DSA is halted and the PP's are directly fetched from the memory so that incomplete MACs without the first step (PP generation) are implemented. If the filter length is M a series of $M/2$ incomplete "dual-MAC"s are implemented by the two CA's. During these $M/2$ cycles also the final adders are halted. When all accumulations are implemented, the two final adders are activated to add the resulting two S and two C terms for two filter outputs. Other HWA accelerator examples are the matrix-vector product accelerator, SAD accelerator, Discrete Cosine Transform (DCT) accelerator, scalar-to matrix product accelerator.

4 Performance Analysis

In this section we present performance analysis of the MAVIP example on Fig. 2. First, VHDL modelling results for the main blocks are summarized in Table 1. The underlying technology is the GS40 0.11 μm standard cell technology from Texas Instruments [13]. Gate count is the size of the network in terms of the number of equivalent gates (2-input NAND). Similar figures for the standard radix-4 Booth-recoded multiplier implemented under the same technology are presented in Table 2. Being smaller in size, MAVIP is able of implementing two multiplication operations in approximately the same time as the radix-4 Booth recoded multiplier. At the same time, MAVIP's functionality is larger than that of the multiplier.

As follows from Table 1, MAVIP may operate with the time period of 3.4ns or with the clock frequency of approximately 295 MHz. Multiplication of an 8x8 matrix with 16-bit unknown entries to a vector with 13-bit components consumes 40 clock cycles or 136 ns at the least. FIR filtering of a 16-bit signal of the length 1024 with eight 13-bit filter coefficients takes totally 4608 clock cycles or approximately 16 000 ns. Computation of the SAD between two 16x16 blocks of 8-bit data takes 34 clock cycles meaning it may be computed as fast as in 116ns at the maximum frequency. Computation of an 8-point DCT (as well as inverse DCT (IDCT)) may be accomplished in 74.8ns and computation of a 2-D DCT of an (8*x*8)-block may be implemented in 1200ns. Subtraction or addition of two (16x16) macroblocks consisting of 8-bit data (motion compensation used in video encoders) may be implemented in 256/8=32 clock cycles or in 110ns. Finally, a series of multiplications of one multiplicand with a pair of multipliers, which is the basic operation of the quantization in video encoders may be implemented with the throughput defined by the period of one clock cycle (3.4ns).

Table 1. VHDL modelling results for the main blocks of the MAVIP

The block	Gate count	Delay (ns)
Array of Adders/DS units	1277	1.57
Selection Block	764	2.97
Compression Array (CA)	715	1.88
Final Adder (CLA)	311	3.4
Total	3067	9.82

Table 2. VHDL modelling results for a radix-4 Booth-recoded multiplier

The block	Gate count	Delay (ns)
PP generation block	1742	1.57
Summation Block	3076	1.88
Total	4822	4.8

Based on above estimates, we next examine the performance of a hypothetic system that incorporates the MAVIP example of Fig. 2 in video encoding. In this example the internal memory of the MAVIP consists of eight 80-bit registers. In our analysis, we assume that the full search over the (31x31) search area is implemented in Motion Estimation (ME) for every macroblock. In a real video encoder most likely much faster algorithm would be used. On the other hand, we assume that the variable-length coding (VLC), the rate control operations as well as other control operations are implemented on the host processor or elsewhere. Assuming that these operations consume 30% of the encoding time and the kernels implemented on the MAVIP consume 70% of the encoding time, one CIF frame is encoded in 0.07sec or in other words 14 frames of CIF resolution in one second (CIF@14 FPS) may be encoded using one MAVIP clocked at maximum frequency. Thus, CIF@30FPS video encoding may be possible to implement on a system involving two MAVIPs. Taking into account the small size of the MAVIP (comparable to the size of one standard multiplier) this is a remarkable result even though the estimates are somewhat theoretical.

Table 3 presents estimates of the numbers of cycles per second (or the required frequency in MHz) consumed by the basic video encoder/decoder kernels in the case of

Table 3. Estimates of frequencies (in MHz) needed to implement video encoder/decoder kernels in QCIF@15FPS on the MAVIP

Kernel	Encode	Decode
Color conversion	2.3	2.3
Motion Estimation (SAD)	11.4	0
Software coprocessor (on host)	5	0
Texture coding	6.3 (DCT&IDCT)	3.2 (IDCT)
Bitstream (on host)	2.7 (coding)	1.6 (decoding)
AC/DC prediction (on host)	0	0.2
Motion compensation	0	0.6
Post-processing (on host)	0	25
Total	**27.7 MHz**	**25 + 7.9 = 32.9 MHz**

implementing QCIF@15FPS encoding/decoding on the system consisting of an ARM processor and one MAVIP. Kernels that cannot be implemented on MAVIP are assumed being implemented on the host (ARM processor). The motion estimation algorithm implemented on MAVIP is considered to be full search over search area of the size (15x15) though typical motion estimation algorithm used in video encoders is usually much less computationally intensive. The total frequency needed for the decoder is shown in two parts, the first part indicating the frequencies needed in actual decoding and the second part indicating the post-processing. As can be seen from Table 3, MAVIP illustrates a competitive performance.

We have analyzed performance of only one small example of MAVIP. There are various ways to scale the performance up. The most straightforward way would be to include several of MAVIP's into one system. Another possibility is to improve balancing between pipelined stages by parallelizing them so that slowest stages are accelerated more than the faster ones (see [10], [11]). This ways, bank of MAVIP's with shared blocks can be derived allowing to increase the throughput with minimum area extension. Yet another possibility to scale the performance up is to derive MAVIP from higher(radix-16) and/or from higher precision ($m > 13$, $n > 16$) multipliers.

Although no simulations of MAVIP were carried out for its power consumption, we can expect low energy consumption since MAVIP is small and should not have higher activity level than conventional designs (moreover, MAVIP should implement less amount of calculations due to reusing potential partial products in many of multiplication operations).

5 Conclusions

A radix-8 multiplier design and its extension to a reconfigurable hardware accelerator for video and image processing, MAVIP, were proposed. MAVIP may be configured to one or another hardware accelerator such as matrix-vector multiplier, FIR filter, or sum-of-absolute-difference accelerator. Being a small device, MAVIP indicates competitive performance in video coding applications.

References

1. Seidel, P.M., McFearin, L.D., Matula, D.W.: Secondary radix recodings for higher radix multipliers. IEEE Trans. Comput. **54** (2005) 111–123
2. Al-Twaijry, H.A., Flynn, M.J.: Technology scaling effects on multipliers. IEEE Trans. Computers **47** (1998) 1201–1215
3. Shah, S., Al-Khalili, A.J., D.Al-Khalili: Secondary radix recodings for higher radix multipliers. In: 12th Int. Conf. On Microelectronics, ICM2000. (2000) 75–80
4. Sam, H., Gupta, A.: Generalized multibit recoding of two's complement binary numbers and its proof with applications in multiplier implementations. IEEE Trans. Computers **39** (1990) 1006–1015
5. Conway, C., Swartzlander Jr., E.: Product select multiplier. In: IEEE 28th Asilomar Conf. On Signals, Systems, and Computers. (1994) 1388–1392
6. Schwarz, E., III, R.A., Sigal, L.: A radix-8 cmos s/390 multiplier. In: 13th IEEE Symposium on Computer Arithmetic. (1997) 2–9
7. Williams, T.: US patent No 4,965,762. Mixed Size Radix Recoded Multiplier (1990)
8. Cherkauer, B., Friedman, E.: A hybrid radix-4/radix-8 low power signed multiplier architecture. IEEE Trans. Circuits and Systems–II, Analog and Digital Signal Processing **44** (1997) 656–659
9. Chen, H.Y., Gai, W.X.: US patent No 5,828,590. Multiplier based on a variable radix multiplier coding (1998)
10. Guevorkian, D., Launiainen, A., Liuha, P., Lappalainen, V.: A family of accelerators for matrix-vector arithmetics based on high-radix multiplier structures. In Pimentel, A., Vassiliadis, S., eds.: Computer Systems: Architectures, Modeling, and Simulation. Volume 3133 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York (2003) 118–127
11. Guevorkian, D., Launiainen, A., Lappalainen, V., Liuha, P., Punkka, K.: A method for designing high-radix multiplier based processing units for multimedia applications. IEEE Trans. Circuits and Sustems for Video Technology (2005) to appear.
12. Guevorkian, D., Launiainen, A., Liuha, P., Lappalainen, V.: Architectures for the sum of absolute differences operation. In: IEEE Workshop on Signal Processing Systems (SIPS 2002). (2002) 57–62
13. Texas Instruments: (2001) http://www-s.ti.com/sc/psheets/srst143/srst143.pdf.

A Scalable Embedded JPEG2000 Architecture

Chunhui Zhang, Yun Long, and Fadi Kurdahi

Department of EECS, University of California,
ET508, zotcode 2625, UCI, Irvine, CA, 92697, USA
{chunhuiz, longy, kurdahi}@ece.uci.edu

Abstract. It takes more than a good tool to shorten the time-to-market window: the scalability of a design also plays an important role in rapid prototyping if it needs to satisfy various demands. The design of JPEG2000 belongs to such cases. As the latest compression standard for still images, JPEG2000 is well tuned for diverse applications, raising different throughput requirements on its composed blocks. In this paper, a scalable embedded JPEG2000 encoder architecture is presented and prototyped onto Xilinx FPGA. The system level design presents dynamic profiling outcomes, proving the necessity of the design for scalability.

1 Introduction

JPEG2000 is the latest compression standard for still images [1]. Due to the adaptations of the discrete wavelet transform (DWT) and the adaptive binary arithmetic coder (Tier-1), JPEG2000 provides a rich set of features not available in its predecessor JPEG. In particular, the core algorithm of the standard addresses some of the shortcomings of baseline JPEG by supporting features like superior low bit-rate performance, lossy to lossless compression, multiple resolution representation, embedded bit-stream and so forth.

Several JPEG2000 encoder architectures have been implemented [2][3][4]. They employed a fixed number of dedicated hardware accelerators for the two compute-intensive blocks, DWT and Tier-1. However, JPEG2000 aims at a broad application scope, thus the relative processing demand on DWT and Tier-1 varies tremendously as the situations alter (e.g. image type as compression ratio). Therefore, such rigid architectures, which restrict the relative throughput between DWT and Tier-1 only optimal at specific conditions, will easily lose the balancing points and even be severely unbalanced under a changed circumstance.

In this paper, a scalable JPEG2000 encoder is presented and prototyped onto Xilinx Virtex-II Pro FPGA. It takes the advantages of Virtex-II Pro's embedded PowerPC RISC core and the flexible on-chip bus structure. The SW/HW (Software/Hardware) partition scheme used is based on our profiling experiments and the hardware accelerators are carefully designed for scalability concern. Beside the performance advantages afforded by the customized hardware cores, the available scalability facilitates the throughput exploration for a given specification. The paper is organized as follows: In Section 2, the proposed JPEG2000 system design is discussed. The two main hardware accelerators for DWT and Tier-1 are presented in Section 3. Section 4 gives the prototyping results together with the scalable performance. The paper is then concluded in Section 5.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 334–343, 2005.

2 JPEG2000 System Design

Before prototyping the encoder onto FPGA, intensive simulations have been done to verify the design. The specification is coded in floating-point C at the beginning and then translated into fixed-point for hardware verification.

2.1 JPEG2000 Overview

A typical JPEG2000 encoder is composed of the fundamental building blocks shown in Fig. 1. The decoding process is symmetric to the encoding but in the reverse direction. This subsection gives a brief overview of the encoding steps and the details are referred to the standard [1].

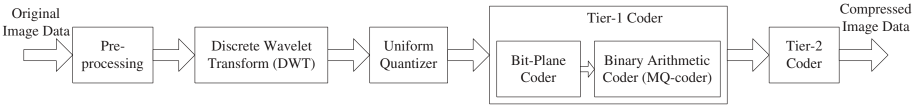


Fig. 1. JPEG2000 fundamental building blocks

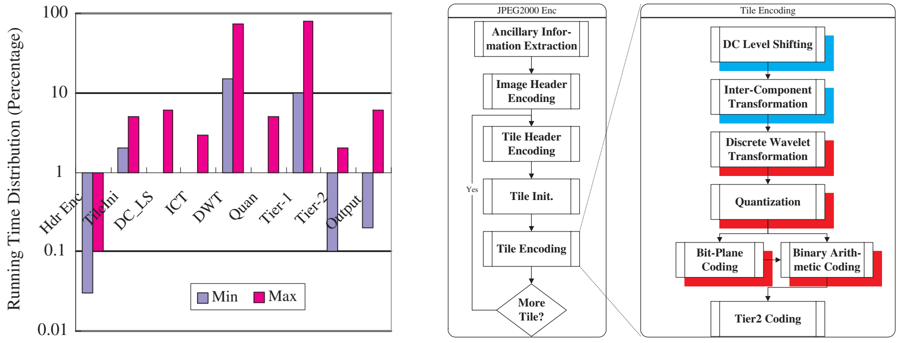
During encoding, the image is partitioned into rectangular and non-overlapping tiles which are processed separately. Next, unsigned sample values are level shifted to make them symmetric around zero. Counting the optional inter-component transform (ICT), those procedures are summarized as pre-processing. The dyadic DWT is applied on the tile repeatedly to de-correlate it into different decomposition levels (resolutions). For lossy compression, the wavelet coefficients are fed to a uniform quantizer with central deadzone.

Each subband is further divided into rectangular blocks namely, code blocks, which are entropy-coded independently. In JPEG2000, entropy coding is two-tiered. Tier-1 is a context-based adaptive arithmetic coder composed of Bit-Plane Coder (BPC) and Binary Arithmetic Coder (MQ-coder). It accepts the quantized wavelet coefficients along with their corresponding probability estimates generated by the BPC, and produces highly compressed codestream. This codestream is carefully organized by Tier-2 coder, constructing a flexible formatted file. New terms and techniques like precinct, packet, tag-tree, and rate control enable features like random access, region of interest coding and scalability.

2.2 Profiling and SW/HW Partitioning

Although C open sources for JPEG 2000 are available, e.g. Jasper [1], almost all of them are software-oriented. Their merits of full specification capture and extra facilities bring complicated design and inferior performance adversely. Therefore, we program our own hardware-oriented JPEG 2000 encoder.

We profile the C implementation of the encoding algorithm on a standard PC (Pentium IV 2.4 GHz, 512M RAM) for rough software estimation and subsequent SW/HW



(a) (b)

Fig. 2. (a)The profiling results; (b) Proposed partitioning schemes

partitioning. Based on the profiling results shown in Fig. 2 (a), two candidate partitions are given in Fig. 2 (b). DWT and Tier-1 coder are computationally, as well as memory, intensive. Therefore, partition 1 allocates DWT, Quantizer and Tier-1 coder into hardware part while the rests run in the host processor in software manner. Partition 2 further assigns DC level shifting and ICT to hardware. By contrast, Tier-2 encoder is computationally inexpensive and showing inferior data locality. Therefore, Tier-2 is appointed to software although it "appears" complicated. Besides performance improvement through hardware customization, the partitioning schemes also decrease communication budgets by confining Tier-2 and the initialization parts in the host processor.

An important phenomena can be noticed via the profiling — the distribution of run time over the blocks varies with large dynamic ranges, prominent for DWT and Tier-1. Actually, Fig. 2 gives two run times, *Min* and *Max*, for each block as the bounds (those optional blocks have zero *Min*). The relative throughput between DWT and Tier-1 alters over ten times in common configuration ranges (e.g. compression ratio from about 2 to 50). When the compression ratio is low (as lossless as the extreme in medical image processing), all information are preserved for Tier-1 processing, whereas most of which could have been eliminated away by the quantizer when the compression ratio becomes high.

2.3 Proposed JPEG2000 Architecture

We propose a system level architecture for JPEG2000 encoding core algorithm targeting toward Xilinx Virtex-II Pro. Fig. 3 shows the diagram based on XC2VP7, a cheap device in Virtex-II Pro family. While the Virtex-II Pro FPGA can be embedded with up to 4 PowerPC cores and 556 multipliers, it is ideal for single-chip embedded system design with scaling up potential. The corresponding decoder can be realized in almost the identical structure by inverting the data path.

Both of the two partition schemes presented in section 2.2 are considered in the architecture. Tier-2 is completed in PowerPC, while DWT and Tier-1 are implemented using hardware-specific design with FPGA logic. The remainder optional blocks (DC_LS,

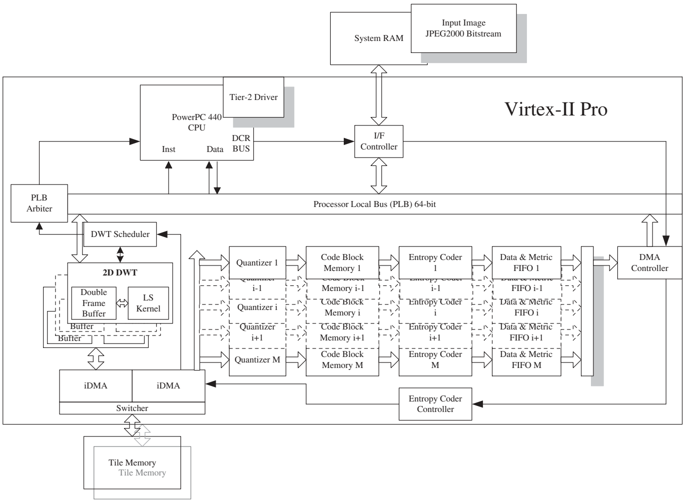


Fig. 3. Proposed JPEG2000 encoder based on Virtex-II Pro XC2VP7

ICT and Quantization) can be either inserted into the data path or run on the PowerPC because of their fine granularity (full parallelism and scalability) and lesser computational requirements. The 64-bit width PLB bus sustains most data communication loads and the DCR bus is utilized for control and parameter-tracing purposes.

The salient feature in the proposed architecture is its scalability. The major blocks, DWT and Tier-1, are configurable with parameters *N* and *M* which represent the numbers of DWT and Tier-1 hardware modules respectively. *N* and *M* are constrained by hardware resources. The corresponding peripherals can be also scaled to sustain the data consuming and delivering. The dedicated hardware accelerators with their scalability will be detailed in Section 3.

2.4 Scheduling Scheme

JPEG2000 fundamental blocks are divided into two parts, DWT part (including inter-component transform, DC level shifting and DWT) and Tier-1 part (Quantization, Tier-1 and Tier-2). Table 1 lists the comparison of four practical scheduling granularity (S1-S4): tile, resolution, subband and precinct. Synchronization here is used as between the two parts. S1 has simple synchronization scheme in contrary with S3 and S4. Furthermore, S3 complicates Tier-2 and S4 conflicts with DWT characteristics. Between the two preferable candidates, S1 shows advantages in most aspects over S2, such as synchronization, memory size and Tier-2 coding. Throughput of S1 and S2 are the same for continuous tiles processing. Smaller latency maybe the only merit introduced by S2. Therefore, S1 is the most appreciated choice which uses tile as the grain size.

The original images, logically divided into tiles, are stored in off-chip memory. Every time, one new tile is transferred to the DWT accelerator through the PLB bus. The decomposed data are stored inside Tile Memory which can contain two tiles and switch in a “ping-pong” way. Concurrently, the previous tile which has just finished DWT pro-

Table 1. Comparison of different scheduling granularity

Scheduling	Tile	Resolution	Subband	Precinct
Granularity	(S1)	(S2)	(S3)	(S4)
Synchronization	Simple	Moderate	Hard	Hardest
Memory size	2T+1P[1]	2T + 1 compressed T	Almost no saving	Almost no saving
Throughput	$1/max\{T_{DWT}, T_{T1}\}$[2]			
Latency	$T_{DWT} + T_{T1}$ $+T_{PT2}$ [3]; or $2 * T_{T1} + T_{PT2}$	Potential Improvement	Smaller than S2	Smallest
T2 coding	Forward	Reverse	Complex	Reverse

cessing starts the procedure of Tier-1 encoding. Tier-1 can be processed in the order of the final bitstream. Thus Tier-2 is encoded in a straightforward way without large amount of buffering.

3 Main Hardware Accelerators

Customized designs for DWT and Tier-1 are primary for our JPEG2000 architecture on Xilinx platform. The C codes of the two blocks are replaced by their VHDL implementations at RTL level. We designed all four DWT engines: default lossless and lossy — LeGal (5, 3) and Daubechies (9, 7); forward and inverse transformations, respectively. Due to the space concern, only forward Daubechies (9, 7) DWT is discussed in this paper.

3.1 Scalable DWT Architecture

Lifting Scheme. The basic principle of lifting scheme is to factorize the polyphase matrix of a wavelet filter into a sequence of alternating upper and lower triangular matrices and a diagonal matrix [5]. Following is one factorization form,

$$A(z) = \begin{bmatrix} H_e(z) & H_o(z) \\ G_e(z) & G_o(z) \end{bmatrix} \tag{1}$$

$$A(z) = \left(\prod_{i=1}^{m} \begin{bmatrix} 1 & s_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ t_i(z) & 1 \end{bmatrix} \right) \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix} \tag{2}$$

where $s_i(z)$ and $t_i(z)$ are Laurent polynomials, which are called prediction and updating operations respectively. Lifting scheme has several advantages, such as the reduction in the number of multiplications (odd-tap filters only) and additions, “in-place” computation, symmetric forward and inverse transform, etc. Although problems of border mirroring, synchronization and programmability caused by lifting scheme have restrained some designers back to the classical convolution way [6], lifting scheme is

[1] T stands for “Tile”, and P stands for “Precinct”

[2] T_{DWT} and T_{T1} mean the processing times of DWT and Tier-1 for one tile

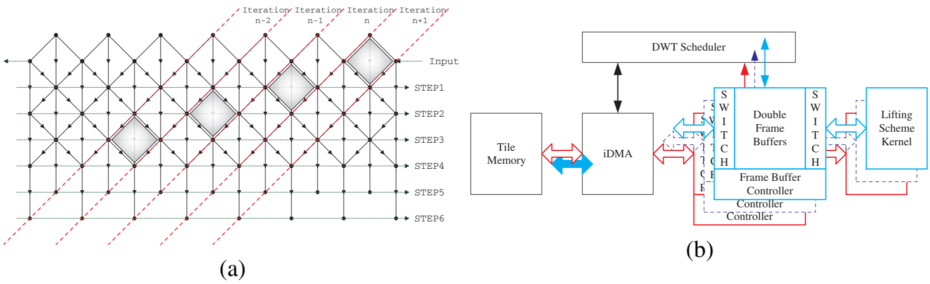
[3] T1 stands for “Tier-1” and T2, “Tier-2”;T_{PT2} is the time for one precinct Tier-2 coding.

widespread [7][8]. We find that the main contribution of “in-place” computation is not to save storage, but to simplify the control. Therefore, lifting scheme is selected in our design for the sake of computation reduction while the “in-place” feature is replaced by “switching frame buffers” for an even simpler control. Although frame buffers are doubled, the whole storage increases less than 1 percent for a 512 by 512 image.

Fixed-Point C Implementation and Precision Analysis. The floating-point C implementation is translated into fixed-point for the precision analysis. We carried out experiments on the peak signal to noise ratio (PSNR) under different filter coefficient word length and concluded that the PSNR saturates at 10-bit. For an 8-bit decompressed image, PSNR is defined as $10log_{10}\frac{255^2}{MSE}$, where MSE refers to the mean squared error between the original image and the reconstructed image. The input signals are shifted left to decrease the normalization errors. EB, Extra Bits, is introduced as the number of shifting bits. Extensive simulations proved that 16-bit signal width with $EB = 5$ satisfy both the required accuracy and the dynamic range.

“Software Pipelined” Lifting Scheme Kernel (LS Kernel). We proposed a “software pipeline” method to implement lifting scheme, referred to [9] for details. Illustrated in Fig. 4 (a), the conventional step by step “zigzag” data flow way is replaced by a single-step scan. Each shadowed diamond is a lifting scheme element (LS element), containing 2 additions and 1 multiplication. A LS Kernel is composed of four LS elements and 2 extra multiplications, delimited by two neighboring dash lines. In order to eliminate data dependencies, software pipeline technique is applied, extracting parallelism among LS kernels thoroughly. Our approach largely improves the data reuse, reducing memory access count about 5 times for Daubechies (9, 7) transform.

Hierarchical Pipelining. The overall DWT architecture is composed of five blocks: iDMA, Frame Buffer, LS Kernel, Scheduler and Tile Memory, shown in Fig. 4 (b). Contrived for image processing, iDMA, or image DMA, is an enhanced DMA engine providing two dimensional addressability. The image is loaded into frame buffer



(a)

(b)

Fig. 4. (a)“Software pipelining” illustration for lifting-based Daubechies (9, 7) Filtering; (b) Proposed DWT block diagram through iDMA and processed line by line. Above the pipelined LS Kernel (computation pipeline), Tile Memory, Frame Buffer and LS Kernel form a higher level of pipeline — task pipeline. Computation pipeline and task pipeline are well balanced. The overall performance is limited by both of them.

Scalability. Due to the symmetry and independence of line processing during one dimensional DWT decomposition at one level, the DWT architecture can be easily scaled up, referred to Fig. 4 (b). For the LS Kernel, duplication can multiple the throughput directly and almost linearly. For the peripherals, throughput can be flexibly controlled by adjusting the data bus width. Such configurability only introduces a little more complexity to the Scheduler.

3.2 Scalable Tier-1 Architecture

Overview. Various architectures of Tier-1 encoder have been proposed. Many of them [4][10] adopt the default mode (or called sequential mode when juxtaposed with parallel mode) of JPEG2000, processing on the code blocks bit-plane by bit-plane. Although default mode has the advantage on the compression rate, it is weak on error resilience and lacks intra-code-block parallelism. To overcome those drawbacks, we use a combination of parallel mode and stripe-causal mode [1] for the purpose of hardware acceleration. As shown in Fig. 5, Tier-1 coder reads the DWT coefficients from Tile Memory and writes the Tier-1 coding result to internal buffer, ready to be fetched by Tier-2 coder. The Tier-1 coder is proposed with full scalability (we use 5 BPCs, each including an MQ-coder, for better illustration). Data Collector is used to collect and organize the output of BPCs for Tier-2 coder. The details of the architecture is referred to [11].

Data Dispatcher. Data Dispatcher is the most control intensive part in the whole design. It is used to load none-all-zero bit-planes to the BPCs. It has an interval state machine working in 3 phases: *initialization*, *programming*, and *program execution*. The state machine is reset at the beginning of each precinct. In the *initialization* phase counters are initialized and starting/ending addresses of a precinct, which are pre-stored by PowerPC, and then loaded. In the *programming* phase, the state machine checks the code blocks in the order that will be used in Tier-2 coding within a precinct, and assigns the bit-planes in those code blocks to the 5 BPCs. During each *program execution*, all 5 BPCs are loaded with a column of data. A program takes 2-5 cycles to execute in different situations.

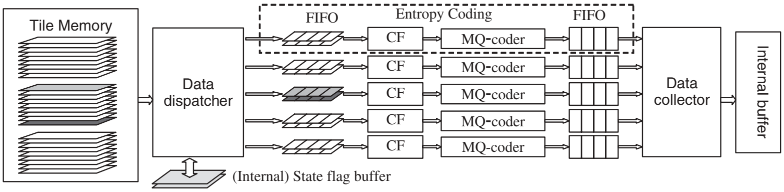


Fig. 5. Tier-1 coder architecture

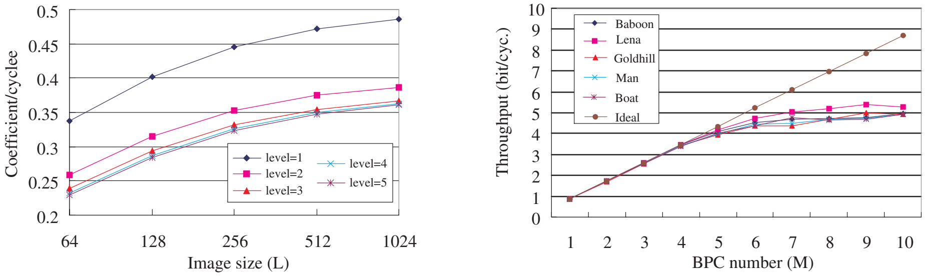
Entropy Coding. The core Tier-1 coding task performs entropy coding, which includes Bit-Plane Coding and Binary Arithmetic Coding, by Context Formatter (CF) and MQ-coder individually, illustrated in Fig. 5. Each BPC acquires the data of one bit-plane from Data Dispatcher column by column. The difficulty of bit-plane coder design is that it scans data in 3 passes, where the latter two passes are dependent on the state bits updated in the earlier passes. Processing data pass by pass requires buffering and reusing the data pairs and those state flags of the whole bit-plane, which demands large internal memory space. Fortunately some research work has been conducted on this problem [10][12]. Our BPC design is inspired by these papers, so that the 3 passes can be combined into one pass, thereby saving memory cost.

4 FPGA Prototyping and Performance

The architecture has been prototyped on Xilinx Virtex-II XC2VP7-7 FPGA. As mentioned before, it can be parameterized with *N* DWT modules and *M* Tier-1 coders with area and performance concerns. The hardware costs are detailed in Table 2. The post-Place&Route shows that the entire system can operate at 130 MHz. XC2VP7 contains 4,928 slices, 44 multipliers and 44 BRAMs, thus it can roughly support up to 2 DWT modules with 6 Tier-1 modules.

Table 2. The main prototyping costs (*N* DWT and *M* Tier-1 modules)

Building blocks	LS Kernel	Frame Buffer	iDMA	Scheduler	Data Dispatcher	BPC+ MQ-Coder	Data Collector	Double buffer
Slice	$167 \times N$	$165 \times N$	$312{+}0.2 \times N$	342	370	$486 \times M$	244	-
Memory and other costs	6 Multi-pliers	8 KB $\times N$	-	-	-	2.8 KB $\times M$	-	1.2 KB $\times M$



(a) (b)

Fig. 6. (a) Throughput of DWT accelerator (*N*=1; test images are square with size *L* by *L*; scalability over *N* is almost linear); (b) Performance scalability of Tier-1 Coder

Fig. 6 (a) demonstrates the experimental evaluation of our DWT architecture in terms of throughput versus decomposition level and image size. As the decomposition level increases, the throughput decreases due to more computational budgets. The throughput is also affected by image size because of the varying relative cost between overhead and computation. For a given circumstance, we can easily determine the number of DWT modules N based on this figure. The measured PSNR is in the range of 48.6-56.52 dB.

The throughput scalability of the proposed Tier-1 architecture is conducted in the sense of BPC numbers M, shown in Fig. 6 (b). The curves go off the ideal course because the Data Dispatcher (section 3.2) is not able to feed that many BPCs. However, such scalability is good enough since 6 BPCs or less satisfy most real-time coding applications. Furthermore, the linearity can be extended by extracting inter-code-block parallelism and Data Dispatcher duplication.

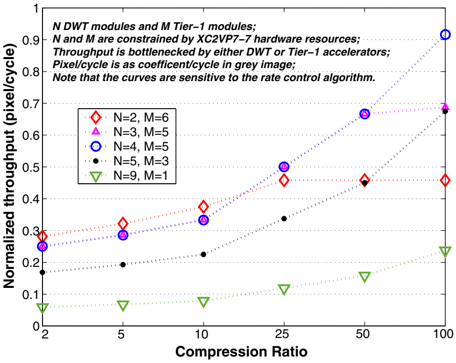


Fig. 7. Throughput exploration by N and M tradeoff (grey image, $level = 5$, $L = 64$)

One of the particular characteristics of FPGA is the in-field reconfigurability. While the number of DWT and Tier-1 modules, (N, M), is constrained by hardware resources, the field-programmability of FPGAs allows us to trade off N versus M within the hardware constraint. Fig. 7 illustrates the throughput exploration of the overall JPEG 2000 encoder by adjusting different (N, M) pair under various compression ratios. Note that the curves are also sensitive to the rate control algorithm and the image itself. In this example, the selection of (N, M) versus the compression ratio CR for optimal throughput are: (2, 6) when $CR \leq 10$; (3, 5), rather than (4, 5) for more efficient power and area despite the same throughput, when $25 \leq CR \leq 50$; and (4, 5) when CR is around 100. In case the throughput constraint is also known, the (N, M) pair which satisfies the real-time processing demand and costs least hardware resource is the optimal tradeoff. For instance, when the throughput constraint is 0.3 pixel/cycle and $CR \geq 25$ in Fig. 7, $(N,\ M) = (5,\ 3)$ is the best choice due to its area efficiency.

A comparison of JPEG 2000 encoders on similar FPGA platforms is given in Table 3. It clearly shows the superiority of our proposal with respect to commercially available IP solutions [2][3]. Actually, the routing costs over half of the critical path delay in our implementation, and the un-optimized FPGA logic also constrains the potential throughput. Generally speaking, the custom IC fabrication can further improve the throughput about two times.

Table 3. Comparison of JPEG 2000 encoders

Architecture	Family	Device	Slices	CLK (MHz)	P(Msample/s)	Image size
Amphion CS6510X2[3]	Virtex-II	-	14,034	40.4	16	1024×1024
CAST JPEG2K_E[2]	Virtex-II Pro	2VP30-7	12,885	107	34.7	256×256
Our implementation[1]	Virtex-II Pro	2VP7-7	4,671	130	60.1	1024×1024

[1] $N = 1, M = 5$; 512 by 512 image; 5 level decomposition; 2VP7-7 has the same speed grad of 2VP30-7;

5 Conclusion

A system level JPEG2000 Part I encoder architecture has been proposed in this paper. It is prototyped on Xilinx technology, showing high performance compared with other designs on similar platforms. The hardware accelerators are carefully designed for scalability concern with adaptive performance.

References

1. JPEG2000 image coding system, ISO/IEC International Standard 15444-1. ITU Recommendation T.800 (2000)
2. JPEG2000 Encoder Core. Cast Inc. (2002)
3. CS6510 JPEG2000 Encoder. Amphion (http://www.amphion.com/cs6510.html)
4. Andra, K., Chakrabarti, C., Acharya, T.: A high-performance JPEG2000 architecture. IEEE CSVT **13** (2003) 209–218
5. Daubechies, I., Sweldens, W.: Factoring wavelet transforms into lifting schemes. Journal of Fourier Anal. Appl. **41** (1998) 247–269
6. Ravasi, M., Tenze, L., Mattavelli, M.: A scalable and programmable architecture for 2-D DWT decoding. IEEE Trans. on Video Tech. **12** (2002) 671–677
7. Andra, K., et al.: A VLSI architecture for lifting-based forward and inverse wavelet transform. IEEE Transactions on Signal Processing **50** (2002) 966–977
8. Chen, C.Y., et al.: A programmable parallel VLSI architecture for 2-D discrete wavelet transform. Journal of VLSI Signal Processing **28** (2001) 151–163
9. Zhang, C., et al.: 'software-pipelined' 2-D discrete wavelet transform with VLSI hierarchical implementation. In: Proc. of RISSP '03. (2003) 148–153
10. Chen, K., Lian, C., Chen, H., Chen, L.: Analysis and architecture design of ebcot for jpeg-2000. In: IEEE ISCAS. (2001) 765–768
11. Long, Y., Zhang, C., Kurdahi, F.: A high-performance parallel mode EBCOT architecture design for JPEG2000. In: Proc. IEEE SOCC. (2004) 213–216
12. Chiang, J., Lin, Y., Hsieh, C.: Efficient pass-parallel architecture for EBCOT in JPEG2000. In: IEEE ISCAS. (2002) 773–776

A Routing Paradigm with Novel Resources Estimation and Routability Models for X-Architecture Based Physical Design*

Yu Hu[1], Tong Jing[1], Xianlong Hong[1], Xiaodong Hu[2], and Guiying Yan[2]

[1] Computer Science and Technology Department, Tsinghua University,
Beijing 100084, P.R.China
jingtong@tsinghua.edu.cn
[2] Institute of Applied Mathematics, Chinese Academy of Sciences,
Beijing 100080, P.R.China

Abstract. The increment of transistors inside one chip has been following Moore's Law. To cope with dense chip design for VLSI systems, a new routing paradigm, called X-Architecture, is introduced. In this paper, we present novel resources estimation and routability models for standard cell global routing in X-Architecture. By using these models, we route the chip with a compensation-based convergent approach, called COCO, in which a random sub-tree growing (RSG) heuristic is used to construct and refine routing trees within several iterations. The router has been implemented and tested on MCNC and ISPD'98 benchmarks and some industrial circuits. The experimental results are compared with two typical existing routers (labyrinth and SSTT). It indicates that our router can reduce the total wire length and overflow more than 10% and 80% on average, respectively.

1 Introduction

As very large scale integrated circuit (VLSI) advances, circuits are getting larger and more complex. Many studies have been focused on high performance integrated circuit (IC) design. However, while designs are implemented based on traditional Manhattan routing architecture, i.e., all interconnects route either horizontally or vertically, the optimization capability is limited due to less routing flexibility. Then, a new routing paradigm allowing interconnects to explore 0°, 45°, 90°, and 135° directions, called X-Architecture, is proposed, which tries to overcome the intrinsic limitation of traditional routing architecture. TX79, a Toshiba micro-processor core, has been replaced and rerouted with the X-Architecture and the liquid routing[1], resulting in an average over 20% wire length reduction and a 20% performance improvement [2].

To cope with the increasing complexity, designers often explore a global router followed by a detailed one. Global routing plays an important role in VLSI physical de-

* This work was supported in part by the NSFC under Grant No.60373012, the SRFDP of China under Grant No.20020003008, and the Hi-Tech Research and Development (863) Program of China under Grant No.2004AA1Z1050.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 344–353, 2005.

sign. Wong [3] proved that it is a NP-hard problem to get the optimal routing solution. Useful algorithms have been proposed focusing on routability [4,5,6,7], timing issue [8,9,10], coupling noise[11,12], and routing tree construction [4,13,14,15,16,17].

Ryan Kastner [5] developed a global router based on maze routing, called labyrinth. Jing et al. [7] presented a congestion reduction global routing algorithm based on search space traversing technology (SSTT), by which large circuits[1] can be routed in a short running time, while keeping good performance.

Some researches have been done on non-Manhattan routing. Chen et al. [18,19] reviewed routing architectures. Non-Manhattan routing is now championed by the X Initiative [20]. However, there is no much literature focusing on X-Architecture based routing approaches. Koh et al. [21] proposed a method and implemented a global router to address modern interconnect problems in Manhattan and non-Manhattan architectures. Agnihotri et al. [22] introduced a layer balance approach for congestion reduction in both Manhattan and non-Manhattan architectures. [23] presented a grid-less routing algorithm to reduce both the total wire length and the number of vias.

Motivated by the recent process of X-Architecture, this paper studies X-Architecture based routing to handle large scale circuits efficiently. The goal of our work is to employ X-Architecture in the global router, by which we can achieve highly routing performance that can hardly be obtained in Manhattan routing. In this paper, we route in X-Architecture with a compensation-based convergent (COCO) approach, in which we use a random sub-tree growing (RSG) heuristic to construct and refine routing trees within several iterations. Using our global router, the design flow mentioned in [1] can be accomplished by performing a liquid routing under the guidance of our global router. Our router is tested on MCNC and ISPD'98 benchmarks and some large industrial circuits, which shows that our router makes substantial improvements on both wire length and routability in the reasonable running time while compared with labyrinth [5,24] and SSTT [7].

The remainder of this paper is organized as follows: in Section 2, resources estimation and routability models in X-Architecture are proposed. Section 3 describes the COCO method for routability analysis. Experimental results, showing routing tree wire length and congestion reduction with MCNC and ISPD'98 benchmarks and industry circuits, are given in Section 4. We conclude this paper in Section 5.

2 Resources Estimation and Routability Models

2.1 Routing Paradigm

[2] mentioned that, a preferred-direction[2] implementation of the X-Architecture is likely to increase the number of vias and can be worse for delay despite the reduction in wire length. To solve this problem, people adopt the gridless octilinear routing technology, called liquid routing, to finish the detailed routing. In our global router, we allow that a routing can use all possible directions in global routing graph (GRG). To guide the

[1] The largest case used has more than 70K nets.

[2] The direction of routing in a particular layer is decided in advance.

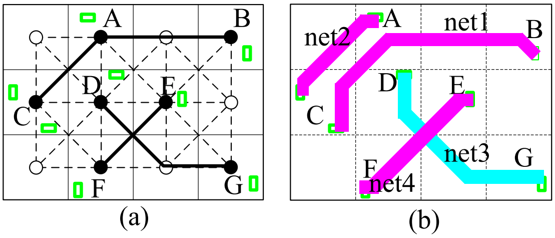


Fig. 1. An example for the global-detailed routing flow

liquid routing, a layer assignment (LA) process should be employed to assign the intersectant segments into different layers. Then the gridless liquid routing can be performed after LA [23].

To explain how to add our global routing results into the design flow in [1], Fig.1 shows an example for routing 4 nets, in which net1 connects pin C and pin B, net2 connects pin C and pin A, net3 connects pin D and pin G, net4 connects pin E and pin F. Fig.1 (a) shows the routing area partition, GRG, and global routing result for these nets and the pins' locations in the chip, in which the green rectangles denote pins. Fig.1 (b) shows the detailed routing result, in which the purple wires are in one layer and the blue ones are in the other layer.

We obtain the global routing results in Fig.1 (a) by using the router presented in this paper. Since net3 and net4 have intersectant segment, DG and EF, they should be assigned to different layers. Then all nets can be connected to their pins following the global routing results with some local adjustments.

2.2 GRG Generation

As mentioned above, we design our global router to guide a gridless liquid detailed router. So it's hard to use the existing GRG generation methods [21, 18] in X-Architecture.

A placement result is needed as an input to our global router. Since the placement algorithms for X-Architecture seems not so mature [2], we utilize a traditional standard cell placer to obtain placement results, which can guarantee the same heights of all rows in standard cell design. We should note that our global router can be performed in any other given placement results after generating the corresponding GRG based on the placement architecture.

According to the height of each row, the entire routing region is divided into a set of rectangular tiles. We prescribe that all non-peripheral tiles (see Fig.2 (a)) should be squares with the same area, which leads the following methods of routing area partition. Assuming the die size of the whole chip is $w \times h$, the vertical distance from top of the chip to the first row is h_t, the one from bottom of the chip to the last row is h_b, and the horizontal distance from the left of the chip to the left most cells is w_l, the number of cells rows is n_{row}. Because all non-peripheral tiles should be squares with the same area, the horizontal width of these tiles is equal to row height, h_{row}. We can see these from Fig.2 (a). The partition parameters can be computed as follows. The number of columns is $n_{col} = \lfloor \frac{w-w_l}{h_{row}} + 2 \rfloor$. The width of the right most tiles is $w_r = w - w_l - (n_{col} - 2) \times h_{row}$.

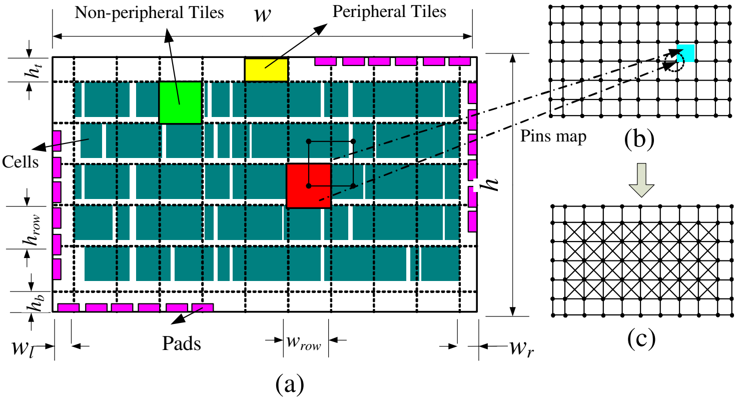


Fig. 2. GRG generation (a) partition of routing region for a standard cell design, (b) the dual graph of (a), (c) the GRG in X-Architecture

After obtaining a partition of the routing area, we generate a dual graph of the partition graph. As shown in Fig.2 (b), each pin is mapped to a vertex corresponding to the tile it is located in. Two adjacent vertices are connected by a horizontal or vertical edge. Then we add some diagonal edges into the dual graph to connect each two diagonal adjacent vertices, except for those in the periphery of the graph, to obtain the GRG in X-Architecture (see Fig.2 (c)). Thus, a net can be specified as a set of vertices in GRG. Then, the problem of routing a net in GRG can be stated as the Steiner tree problem of specified vertices in GRG.

2.3 Routing Resources Assignment (RA) Model

To make sure of the legal via locations in traditional routing algorithms, it's necessary to align the diagonal routing grids to intersect with the Manhattan routing grids below them since each layer employs a preferred routing direction. Then, the result is in an unacceptable waste of wiring resources [2]. In our routing paradigm, routing can explore all possible directions in a layer. So we can compute routing resources according to pitches and routing area, then a liquid detailed router can be used to accomplish the routing process based on our global routing results as we mentioned above. We can use the following method to assign the routing resources.

A positive number c, called edge capacity, is assigned to each edge in GRG. Edge capacity indicates the number of available tracks between the corresponding tiles. Firstly, we compute the number of tracks assigned to each rectilinear edge based on wire widths, pitches, layers and routing area in traditional way. Then we remain half of the capacity of each edge, and reassign the other half to diagonal edges. The Fig.3 shows an example of this strategy.

Fig.3 (a) shows the initial capacity in each edge in rectilinear GRG is 20. We keep the capacities in peripheral edges (not labelled in the Fig.) constant and transfer part of capacities in non-peripheral rectilinear edges to diagonal ones. Then, we half reduce the capacity of each rectilinear edge, which results in capacity 10 in each edge (see

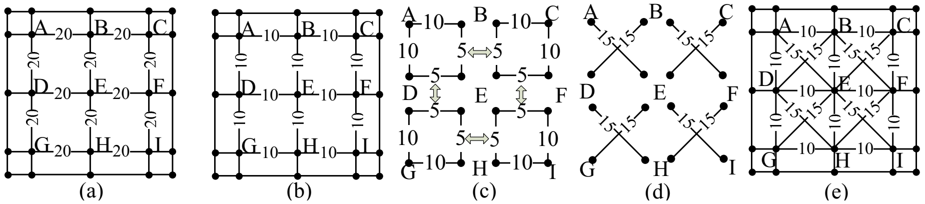


Fig. 3. Resource assignment process

Fig.3 (b)). After that, we assign the other half of capacities into diagonal edges. We consider each non-peripheral square in GRG separately. For example, in square (A, B, E, D), edge (B, E) is shared with square (B, C, F, E), so we half partition the capacity in edge (B, E) for the two separated squares sharing it (see Fig.3 (c)). We assign the total capacities in each square into the two diagonals in it evenly (see Fig.3 (d)). For example, the total capacities in square (A, B, E, D) is 30, so we assign 15 into diagonal edge (A, E) and (B, D), respectively. The final assignment is shown in Fig.3 (e).

2.4 Routability-Aware Global Routing Model

In this paper, both total wire length and total overflow of all edges are considered in our objective. Some basic definitions and formulations are given as follows.

The total wire length is: $L = \sum_{i=1}^{N_n} \sum_{k=1}^{N_e} S_{ik} \cdot l_k$, where N_n is the number of nets, N_e is the number of edges, S_{ik} is equal to 1 if edge e_k is used by the Steiner tree of net n_i, otherwise it's equal to 0, and l_k is the length of edge e_k. The usage of edge e_k is: $u_{e_k} = \sum_{i=1}^{N_n} S_{ik}$, the overflow of edge e_k is:

$$eof_{e_k} = \begin{cases} u_{e_k} - c_{e_k}, & \text{if } c_{e_k} < u_{e_k}; \\ 0, & \text{otherwise,} \end{cases} \tag{1}$$

where c_{e_k} is the capacity of edge e_k. The total overflow of all edges is: $tof = \sum_{k=1}^{N_e} eof_{e_k}$, then, the global routing problem can be described as follows. Find S_{ik} to

$$\text{Minimize } COST = L \times (tof^{\beta} + \varepsilon). \tag{2}$$

where β is a constant, and ε is a small real constant.

3 COCO Method for Routability Analysis

3.1 RSG for Octilinear Steiner Tree Construction

We find that the existing tree algorithms [25, 26, 13] can't be used directly in our graph-based router because the produced solution could introduce some Steiner points not defined in the routing graph. So we present a random sub-trees growing heuristic (RSG) and integrate it into the global router to construct the initial routing tree for each net and to generate different topologies for each net through the routing process.

In our RSG, we denote the given GRG as $G = (V,E)$ and the terminal set as N. The distance between two vertices is defined as the same as in [13]. We use a random subtrees growing strategy to obtain a Steiner minimal tree in the graph G. The pseudo-code of RSG is shown in Algorithm1.

Algorithm 1 Graph-based Octilinear Steiner Minimal Tree
Input: Graph G and terminal set T
Output: An octilinear steiner minimal tree in G

```
for each terminal $p \in$N do
    $t_n \leftarrow$ new subtree based on $p$
    $t_n.GP \leftarrow p$
end for
while the number of subtrees $> 1$ do
    $t \leftarrow$ select a subtree randomly
    $t$ grows from $t.GP$ to vertex $v$ based on Eq. (3)
    if the vertex $v \in$ subtree $t'$ then
        $t_{new} \leftarrow$ merging $t$ and $t'$
        Compute the location of $t_{new}.GP$
    end if
end while
Prune all non-terminal leaves in the last subtree ($tree$)
return $tree$
```

Given a subtree t, whose GP is p, the vertex v it'll grow to is defined as follows:

$$v = \min_{k} \eta^{t}_{p,k}, \tag{3}$$

where k is the neighbor vertex with p, and k is not covered by subtree t. The $\eta^{t}_{p,k}$ is defined as follows:

$$\eta^{t}_{p,k} = len(p,k) + \gamma \cdot \psi^{t}_{k}, \tag{4}$$

where $len(p,k)$ is the length of edge (p,k) in graph G, γ is a constant, and ψ^{t}_{k} is the shortest distance (in X-Architecture) from vertex p to all the vertices covered by other subtrees, which makes the current subtree t grows towards others as quickly as possible.

We noted that a sub-tree will be selected randomly in each iteration. Hence the solution produced by our RSG could vary in different runs. Tested by running randomly generated cases for tens of times, RSG can produce various topologies while keeping the stable and high performance in wire length. This characteristic of RSG will be useful in our global router. Fig.4 shows different topologies for IBM02-net 106 (22 pins), generated by RSG in three runs.

3.2 COCO Method

In this section, we describe our compensation-based convergent (COCO) method for routability analysis, which tries to achieve a global minimal solution instead of getting trapped in local minima by finding a near optimal trade-off between total wire length and total overflow. We use RSG to construct initial trees for all nets. Then several iterations are needed to refine the solution guided by the objective in Eq.(2).

In each iteration, we compute the overflow of each edge in the global routing graph by Eq. (1) and get a set of congested nets, called N_{cong}. Then we randomly select a sub set of N_{cong}, named N_{ran}, according to a probability p_{θ}. The routing resources used by nets in N_{ran} are given back to the edges. The nets in N_{ran} will be rerouted simultaneously. Obviously, there is no net ordering problem among nets in N_{ran}.

To construct a routing tree for each net in N_{ran}, we use the variation of our RSG algorithm. A weight w_e is given in an edge e in the global routing graph, and Eq. (4) is rewritten as follows:

$$\eta_{p,k}^{t} = \log\left(len(p,k) + \gamma \cdot \psi_p^t\right) + \mu \cdot \log w_e, \tag{5}$$

where μ is a constant, and w_e could be w_{1e}, w_{2e} or w_{3e}, which will be defined later.

To avoid trapping into a local minimization, we change rules for weighting an edge in different iterations. We design three kinds of rules, which have different capabilities to reduce total overflow and total wire length. The definition of these rules is shown as follows:

$$w_{1e} = 1, \quad w_{2e} = \begin{cases} \infty, \text{ if } c_{e_k} \leq u_{e_k} \\ 1, \text{ otherwise} \end{cases}, \quad w_{3e} = \begin{cases} 20 \times u_{e_k}/c_{e_k}, \text{ if } c_{e_k} \leq u_{e_k} \\ (u_{e_k} + \varepsilon)/c_{e_k}, \text{ otherwise.} \end{cases} \tag{6}$$

where ε is a small real number that validates the equation while u_{e_k} is 0.

If the w_1 is used in a certain iteration, nets in N_{ran} will be rerouted based on wire length minimization. In the iterations that w_2 or w_3 is used, the wire length of a net could be increased for detouring the congested edges. Under w_1 rule, a net will be rerouted as a Steiner minimal tree. By doing so, the total wire length can be reduced, and the used routing resources can be expected to reduced too, because less GRG edges are used by a net with a shorter length. For those nets without detours, as mentioned in Section3.1, the topologies could be expected to change after rerouting, which indicates the possibility of climbing up from a local minimization.

If the w_2 is used, a routing tree will try to avoiding the congested edges, because of their ∞ weights. In the RSG algorithm, an subtree will avoid choosing an edge if the it's congested, otherwise, it'll choose the shortest path to grow to. So this rule is a tradeoff between congestion and wire length.

If the w_3 is used, a routing tree will grew mainly based on the congestion map. As shown in Eq. (6), the congested edges are multiplied by 20, which makes weights increase sharply in these edges. The wire length information should also be added to consideration because the routing tree with too many detours will lead to the more serious congestion problem.

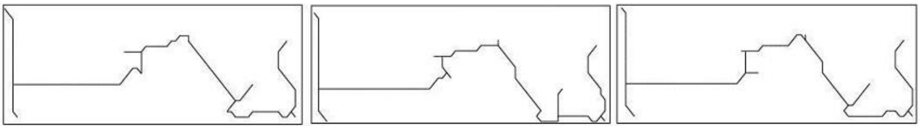


Fig. 4. Different topologies for IBM02-net106 generated by RSG in three runs

The w_3 weight could reduce congestion efficiently while expending longer wire length. On the other hand, the w_1 weight could compensate the wire length increase caused by weight w_3, while producing more congested edges. And the w_2 rule can be seemed as a tradeoff between w_1 and w_3, but it has a limitation of reducing congestion and always leads to a local minimization. In the experiments, we found that we can always get a near optimal solution by employing these three rules as a appropriate sequence.

We run several iterations using w_2 after obtaining the initial routing, until the solution (using the Eq. (2) to evaluate our solution) can not be improved any more, then the weight rule alternates between w_1 and w_3 in the following iterations, until no improvement can be produced. In each iteration, the upper bound of total overflow is given based on the previous congestion map. The decrease of overflows leads to the decrease of congested nets and the number of elements in set N_{ran}, which guarantee the convergence of the algorithm.

4 Experimental Results

We implemented our X-Architecture based global router in C Language on a Sun-fire v880 workstation. We tested our router and compare to SSTT[3] [7] and Labyrithn [11, 24] (the benchmarks are also from the respective literature).

Table 1. Wire length of final routing (COCO vs. SSTT)

Circuit	COCO (μm)	SSTT (μm)	Imp. (%)
C2	5.05e+05	5.45e+05	7.34
C5	1.09e+06	1.26e+06	13.5
C7	1.53e+06	1.83e+06	16.4
sl3207	9.52e+06	9.74e+06	2.26
avq	9.18e+06	1.08e+07	15.0
u100	3.69e+07	4.12e+07	10.4
ut	9.12e+07	1.15e+08	20.7
ucnt500	2.83e+08	3.02e+08	6.29
u05	1.02e+11	1.19e+11	14.3
u11	1.72e+10	1.97e+10	12.7
Average	1.20e+10	1.39e+10	13.7

Table 2. Total overflow and running time (COCO vs. SSTT)

Circuit	COCO		SSTT		Imp
	ovf	cpu	ovf	cpu	(%)
C2	9	0	18	0	50.0
C5	2	5	50	0	96.0
C7	12	9	46	5	73.9
sl3207	1	57	23	7	95.7
avq	1	143	2	9	50.0
u100	7	0	53	1	86.8
ut	1	0	2	1	50.0
ucnt500	2	5	13	3	84.6
u05	6	739	43	574	86.1
u11	5	1890	15	1942	66.7
Average	5	285	26.5	254	82.6

Comparisons for group 1 on total wire length of final routing are shown in Tab.1. Column *Imp.* shows the wire length reduction percentage of our router (denoted by COCO) beyond SSTT.

We can see that, our router optimizes the wire length beyond SSTT in all test cases, and gets an average 13.7% reduction of total wire length of final routing. This result

[3] Since both SSTT and our router have strong capabilities of congestion reduction, we reduce the capacity of each edge in the same proportion until our router fails to achieve a completed routing. Then SSTT is tested with these shrunken circuits.

Table 3. Wire length of final routing (COCO vs. labyrinth)

Circuits	COCO	Labyrinth	Imp. (%)
ibm01	69575	76517	9.07
ibm02	188691	204734	7.84
ibm03	158837	185194	14.2
ibm04	187443	196920	4.81
ibm05	42417	689671	38.5
ibm06	314082	346137	9.26
ibm07	391289	449213	12. 9
ibm08	440612	469666	6.19
ibm09	467727	481176	2.80
ibm10	685521	679606	-0.87
Average	294619	377883	10.2

Table 4. Total overflow and running time (COCO vs. Labyrinth)

Circuits	COCO		Labyrinth		Imp.
	ovf	cpu	ovf	cpu	(%)
ibm01	60	10	398	72	84.9
ibm02	0	30	492	123	100
ibm03	0	26	209	148	100
ibm04	385	42	882	278	56.4
ibm05	0	57	251	233	100
ibm06	0	53	834	171	100
ibm07	0	66	697	381	100
ibm08	1	73	665	364	99.9
ibm09	1	82	505	553	99.9
ibm10	661	115	588	692	-12.4
Average	110	55	552	302	82.9

is a little conservative because the shrink of the chip increases the detours in the final routing.

Comparisons for group 2 on total wire length of final routing are shown in Tab.3. We can see that our router optimizes the wire length beyond labyrinth in most test cases, and gets an average 10.2% reduction of total wire length.

Tab.2/Tab.4 shows the comparison between our router and SSTT/labyrinth on total overflow (denoted by ovf) and running time. Column *Imp.* shows the reduction percentage of total overflow of our router beyond SSTT/labyrinth. We can find that our router optimizes both the total overflow beyond SSTT/labyrinth in almost all test cases, and gets an average 82% reduction of total overflow.

5 Conclusions

This paper presents a novel routing resource estimation and routability analysis models. Based on these models, we design and implement an X-Architecture based global router for standard cell design. Tested on two groups of circuits, our router achieved an average over 10% reduction of wire length and over 80% reduction of total overflow, when compared with a typical Manhattan global router SSTT. Since the RSG is based on graph, our router could be easily transplanted to other architectures.

As future work, we will consider timing and coupling issues in our router.

References

1. Mitsuhashi, T., Someha, K.: Performance-oriented layout design, pervasive use of diagonal interconnects reduces wire-length. Design Wave Magazine (2001) 59–64
2. Teig, S.: The x architecture: Not your father's diagonal wiring. In: Proc. of SLIP. (2002) 33–37
3. Sarrafzadeh, M., Wong, C.K.: An Introduction to VLSI Physical Design. McGraw Hill, USA (1996)

4. Bozorgzadeh, E., Kastner, R., Sarrafzadeh, M.: Creating and exploiting flexibility in rectilinear steiner trees. IEEE trans. on CAD **22** (2003) 605–615
5. Kastner, R., Bozorgzadeh, E., Sarrafzadeh, M.: Predictable routing. In: Proc. of ICCAD. (2000) 110–114
6. Hadsell, R.T., Madden, P.H.: Improved global routing through congestion estimation. In: Proc. of the DAC. (2003)
7. Jing, T., Hong, X., Bao, H., Xu, J., Gu, J.: Sstt: Efficient local search for gsi global routing. J. of Compute Science and Technology **18** (2003) 632–639
8. Jing, T., Hong, X., Xu, J., Bao, H., Cheng, C.K., Gu, J.: Utaco: A unified timing and congestion optimization algorithm for standard cell global routing. IEEE Trans. on CAD (2004) 358–365
9. Lin, S.P., Chang, Y.W.: A novel framework for multilevel routing considering routability and performance. In: Proc. of ICCAD. (2002) 44–50
10. Hong, X.L., Jing, T., Xu, J.Y., Bao, H.Y., Gu, J.: Cnb: A critical-network-based timing optimization method for standard cell global routing. J. of Computer Science and Technology (2003) 732–738
11. Kastner, R., Bozorgzadeh, E., Sarrafzadeh, M.: An exact algorithm for coupling-free routing. In: Proc. of ISPD. (2001) 10–15
12. Xu, J., Hong, X., Jing, T., Cai, Y., Gu, J.: A novel timing-driven global routing algorithm considering coupling effects for high performance circuit design. IEICE Trans. on Fundamentals of ECCS (2003) 3158–3167
13. Zhu, Q., Zhou, H., Jing, T., Hong, X.L., Yang, Y.: Spanning graph based non-rectilinear steiner tree algorithms. IEEE Trans. on CAD (2005)
14. Wang, Y., Hong, X., Jing, T., Yang, Y., Hu, X., Yan, G.: Spanning graph based non-rectilinear steiner tree algorithms. LNCS3256 (2004) 442–452
15. Alpert, C.J., Hrkic, M., Hu, J., Kahng, A.B.: Buffered steiner tree for difficult instances. In: Proc. of ISPD. (2001) 4–9
16. Xu, J., Hong, X., Jing, T., Cai, Y., Gu, J.: An efficient hierarchical timing-driven steiner tree algorithm for global routing. INTEGRATION, the VLSI J. (2003) 69–84
17. Hrkic, M., Lillis, J.: S-tree: A technique for buffered routing tree synthesis. In: Proc. of DAC. (2002) 578–583
18. Chen, H., Yao, B., Zhou, F., Cheng, C.K.: Physical planning of on-chip interconnect architecture. In: Proc. of ICCD. (2002) 30–35
19. Chen, H., Zhou, F., Cheng, C.K.: The y-architecture: yet another on-chip interconnect solution. In: Proc. of ASP-DAC. (2003) 840–846
20. Organization, T.X.I.: http://www.xinitiative.org (2005)
21. Koh, C.K., Madden, P.H.: Manhattan or non-manhattan? a study of alternative vlsi routing architectures. In: Proc. of the GLSVLSI. (2000) 47–52
22. Agnihotri, A.R., Madden, P.H.: Congestion reduction in traditional and new routing architectures. In: Proc. of GLSVLSI. (2003) 28–29
23. Paluszewski, M., Winter, P., Zachariasen, M.: A new paradigm for general architecture routing. In: Proc. of GLSVLSI. (2004) 26–28
24. Labyrinth. http://www.ece.ucsb.edu/ kastner/labyrinth/ (2004)
25. Coulston, C.S.: Constructing exact octagonal steiner minimal trees. In: Proc. of GLSVLSI. (2003) 1–6
26. Kahng, A., Mandoiu, I., Zelikovsky, A.: Highly scalable algorithms for rectilinear and octilinear steiner trees. In: Proc. of ASPDAC. (2003) 827–833

Benchmarking Mesh and Hierarchical Bus Networks in System-on-Chip Context

Erno Salminen[1], Tero Kangas[1], Jouni Riihimäki[2], Vesa Lahtinen[3],
Kimmo Kuusilinna[3], and Timo D. Hämäläinen[1]

[1] Tampere University of Technology, P.O. Box 553, FIN-33101 Tampere, Finland
erno.salminen@tut.fi
[2] Nokia Technology Platforms, P.O. Box 88, FIN-33721 Tampere, Finland
[3] Nokia Research Center, P.O. Box 100, FIN-33721 Tampere, Finland

Abstract. A simulation-based comparison scheme for on-chip communication networks is presented. Performance of the network depends heavily on the application and therefore several test cases are required. In this paper, generic synthesizable 2-dimensional mesh and hierarchical bus, which is an extended version of a single bus, are benchmarked in a SoC context with five parameterizable test cases. The results show that the hierarchical bus offers a good performance and area trade-off. In the presented test cases, a 2-dimensional mesh offers a speedup of 1.1x - 3.3x over hierarchical bus, but the area overhead is of 2.3x - 3.4x, which is larger than performance improvement.

1 Introduction

The increasing complexity of digital systems has forced the adoption of modular design techniques and the re-use of pre-designed and pre-verified components in the design process of System-on-Chips (SoC). Due to the increasing number of connected components, the communication and interconnect wiring are becoming a serious problem [1][2][3]. Several architecture and circuit-level alternatives have been proposed to solve this but proposals frequently only deal with the theoretical limitations of the communication networks. However, the practical limitations and requirements for the networks are affected by system-level issues and the data transfer distributions of the targeted applications. This paper presents a study of hierarchical bus and mesh networks and examines the effect of transfer distributions in benchmarking SoCs. Furthermore, the area cost of network area is examined. The following Section briefly introduces the related work done in the field. The utilized networks are presented in Section 3 and the transfer patterns in Section 4. Section 5 examines the area costs of the implementations based on synthesis results and describes the execution time comparison. In Section 6, the conclusions of the work are given.

2 Related Work

The optimal SoC communication network (network-on-chip, NoC) has been subject to debate over the last few years. Traditionally, the SoC networks are based on circuit-

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 354–363, 2005.

switched techniques, such as bus-based networks [4], crossbars [5], and 2-dimensional meshes [6][7]. Several packet-switched network topologies have also been proposed, such as fat trees [8], 2-dimensional meshes [9][10], and rings [11]. Networks are often compared by the theoretical maximum transfer capabilities. This is straightforward but the results seldom reflect the performance of real applications accurately, i.e. the order of performance values may be defined but not their ratio. However, building actual applications is laborious only for comparison purposes and their simulation is time-consuming. Furthermore, the execution time of an application is often dependent on input data.

Communication generators provide flexibility and accelerate simulation with the cost of reduced accuracy. They use transfer patterns, or profiles, that resemble the external behavior of real applications. In statistical generators [12][13], the communication profile is often independent of the profiles of other agents (processing elements) that are connected to the network. Statistical methods are suitable for analyzing and optimizing the network when the system architecture and application mapping are fixed. Transfer dependent methods [14][15][16], in contrast, enable architecture exploration including component allocation, application mapping, and communication scheduling. Transfer dependence refers to a situation in which an agent cannot proceed before it has received certain data from other agent(s).

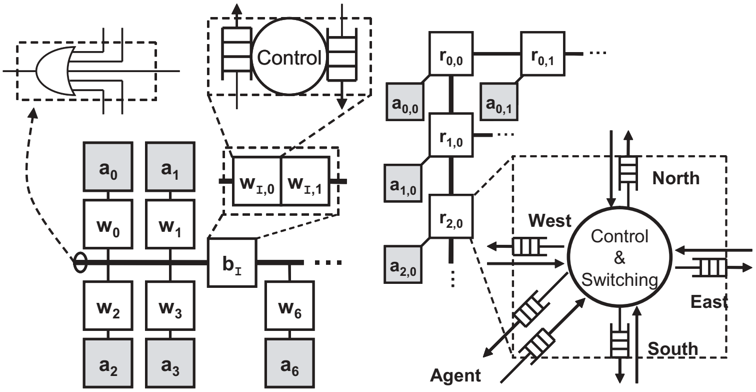
Table 1. NoC comparisons in literature

Ref.	Topologies	Test cases	Criteria
[6]	mesh, (hier.) bus	5 applications	ex.time, throughput
[7]	mesh, bus	1 random, 1 statistical	offered load, blocking
[8]	fat-tree, bus	total exchange, 1 statistical	ex.time, latency, saturation
[10]	mesh	3 statistical	ex.time, latency, area
[11]	ring	1 application	ex.time, area
[12]	(hier.) bus, ring	tens of statistical	throughput
[13]	mesh	1 statistical	ex.time, utilization, power
[16]	mesh, bus, tree	3 transfer dependent	ex.time
[17]	mesh, bus	total exchange	ex.time, max freq.
[18]	(hier.) mesh, multibus	(at last) 3 applications	energy
this work	mesh, (hier.) bus	5 transfer dependent	ex.time, area

Table 1 lists some comparisons of different networks. All utilize simulation, except [17] that uses mathematical models. Three reported studies used real applications while seven used synthetic methods, only one of which included transfer dependence. Total exchange (also called pooling) means that all agents send and receive data with all other agents. A single bus is a viable solution for systems with a limited number of agents and bandwidth requirements because of its simplicity and the numerous legacy implementations [8][16][12]. Many comparisons examine single buses or multiple parallel buses but omit the more versatile hierarchical bus structures. All listed comparisons neglect the cost of network in terms of area, except [10][11] which compare only different versions of single topology and analytical work in [17]. However, area of NoC routers varies greatly; between 4-700 kilogates [10][9], and therefore area should be included in comparison. A single test case is not enough to have reliable performance estimates, but a set of benchmarks is needed. This work utilizes synthetic, transfer-dependent test cases for fair benchmarking of hierarchical bus and 2-D mesh. Currently there are no commonly accepted benchmark sets for SoC networks, but the work presented in this paper could be proposed as part of such set.

3 Hierarchical Bus and Mesh

To allow fair comparison, bus, hierachical bus, and 2-D mesh are compared for this study by using generic, synthesizable building blocks. All networks utilize the same agent interface, transfer data in fixed-size packets, and utilize store-and-forward routing. Data arrives always in-order.



(a) Hierarchical bus (b) Mesh

Fig. 1. Network implementations

The utilization of the single bus architecture is usually limited to systems with less than 10-20 agents due to the limited bandwidth and the problems induced by the required long signal lines. The traditional bus scheme can be extended to a hierarchical bus scheme by using bridges (marked with $b_{I,i}$) to connect several bus segments as shown in Fig 1(a). Agents (a_i) are connected to bus segments via wrappers (w_i). In this case, the number of signal lines and the operating frequency are the same in each segment. Hierarchical bus architecture used in this paper is built as a chain of bus segments, although a tree like structure could also be used. The implemented bus wrapper is a fairly simple device with two FIFO buffers and a control unit for arbitration. Arbitration is based on a distributed round-robin scheme where the ownership is passed to the next agent after each transmitted packet. Bridge components were implemented by connecting two wrappers together. The bus signal resolution is implemented with an OR-based structure. The problems inherent in long bus signal wires are solved by grouping only a limited number of agents, in this case four, in each bus segment.

Table 2. The number of communication links in networks

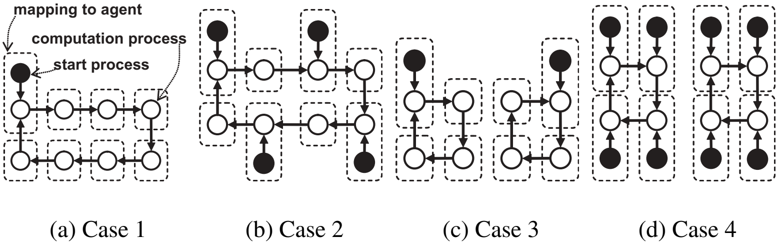
	Number of agents				
Network	4	16	36	64	N
Single bus	1	1	1	1	1
Hier. bus	1	4	9	16	$N/4$
Mesh	8	48	120	224	$4(N-N^{0.5})$

A packet-switched 2-dimensional 4-way mesh used in this study is depicted in Fig 1(b). The agents are connected to an array where router elements (r_i) handle the storing and forwarding of data. Router comprises of a control unit taking care of switching and routing, and a FIFO buffer for each direction (North, East, South, West, and two for agent). Routers implement a simple dimension-order routing scheme where the transfers are first directed to the correct row and then to the requested column. The number of communication links L of networks are listed in Table 2. A single bus has only one communication link, whereas in a hierarchical bus there are as many links as there are bus segments. In mesh, there are 4 unidirectional links in each router except on those residing on edges.

4 Test Cases

The presented communication networks are compared using synthetic benchmarks that are generated to represent characteristic application properties, such as sequential/parallel behavior, communication/computation intensiveness, and spatial traffic distribution. The test cases are executed in a simulation environment called Transaction Generator (TG) [15] that is independent of the network and runs application descriptions based on the Kahn process network model [19]. Each process can be either waiting for data, reading data, processing, writing data, or finished. TG notably accelerates the simulation compared to HW/SW co-simulation with multiple instruction set simulators. At the same time, the timing error is less than 15% w.r.t. real application.

Fig 2 shows the process network graphs of the benchmarks for 8 agents. The white nodes depict computation processes that can have any arbitrary processing times of P clock cycles. The edges represent data transmissions with length of D words. Computation at a node cannot start until at least one of the arriving transfers has completed (transfer dependence). The start processes, marked with black nodes, have $P = 1$ and $D = 1$ and are executed only once to trigger computation processes. By changing P and D, the application model in TG can be made more computation or communication intensive. Both P and D can be varied randomly within a user-defined range. All these benchmarks are scaled with N so that there is one computation process and variable number of start processes per agent. The dashed lines describe a mapping of the process graph onto eight agents. For simplicity, a 1-to-1 mapping was utilized. However, process graphs are totally independent of the hardware architecture so other mappings can be easily explored.



(a) Case 1 (b) Case 2 (c) Case 3 (d) Case 4

Fig. 2. Test cases used in comparison

The first benchmark, case 1, (Fig 2(a)) resembles a sequential data flow application having only one start process. Case 2 (Fig 2(b)) is partly sequential and partly parallel in nature having $N/2$ start processes. Case 3 (Fig 2(c)) presents an application where the transfers are sequential as in case 1 but in hierarchical clusters of four processes so communication is very localized. Case 4 (Fig 2(d)) has processes in a group of four transmitting data in parallel to each other. The difference to case 3 is that there are four simultaneous data transfers in each group of four agents. Cases 2 and 4 can also be thought as pipelined versions of cases 1 and 3, respectively. In addition, case 5 combines the cases 1-4 into one simulation to represent heterogeneous behavior. In case 5, cases 1-4 are run together so that each agent executes one computation process from each case. The mutual order of cases is not specified. For example, all cases have start and computation processes grouped together in the top left corner (cf. Fig 2(a)-2(d)) and they all are mapped to first agent in case 5.

The total execution time of an application is a sum of computation time and communication time. It can be estimated in a heuristic fashion for these graphs as

$$t_{\text{tot}} = \frac{\sum P_i}{min(N,S)} + \frac{\sum (D_i * k)}{min(N,L,S)} \ , \qquad \text{[clock cycles]} \tag{1}$$

where

$$k = \frac{payload + header + arbitration}{payload} \tag{2}$$

is an implementation specific factor that is explained later. The divider in (1) defines the achievable parallelism. The parallelism of an application is defined here as maximum number of parallel transfers and active computation processes and it equals the number of start processes (S) in these cases. If there are less agents (N) or communication links (L) than start processes (S), the maximum parallelism of the application cannot be achieved. The maximum number of initiated transfers per clock cycle cannot exceed the number of agents in any network. For example, case 1 is a sequential application having only one start process and utilizes only one processor or communication link at a time. Adding more communication links should not speed up the application at all. The number of start processes is shown in Table 3.

Table 3. The number of start processes in test cases

	Number of agents				
Test case	4	16	36	64	N
1	1	1	1	1	1
2	2	8	18	32	$N/2$
3	1	4	9	16	$N/4$
4	4	16	36	64	N
5	8	29	64	113	$7N/4+1$

The factor k, caused by arbitration and the overhead from packet headers, is calculated with (2). It is defined as the number of clock cycles needed to transfer one packet divided by the amount of transferred payload data. Ideally k would be one. Packet and header sizes are expressed as multiples of the word size, because one word can be transferred in one clock cycle. Using distributed arbitration, each agent in a bus segment has to wait a whole round-robin iteration between consecutive packets. However, there can be S agents active in each round which reduces the overall arbitration delay. The routing algorithm of a mesh checks one input each clock cycle for new transfers, thus, on average $5/2$ clock cycles are needed for routing. The term arbitration is assumed to be $(N-1)/S$ for bus, 6 for hierachical bus, and 2.5 for mesh. Still, (1) does not take data dependencies into account and assumes uniform mapping of processes. Estimate for case 5 is a sum of individual estimates for cases 1-4.

5 Synthesis and Simulation Results

The communication networks were implemented using RTL VHDL and synthesized using a 0.18 μm CMOS technology. The packet size was set to eight 32-bit words. Since packets have a three-word header and eight-word payload data, the required buffer size is $(3+8)*32$ bits in all the internal network buffers. The resulting network logic areas in kilogates are depicted in Table 4. The difference is mainly due to buffers: bus wrappers have two buffers, bus bridges four, and routers have six buffers.

Table 4. Logic areas in kilogates of network implementations

	Number of agents			
Network	4	16	36	64
Single bus	30	119	269	479
Hier. bus	30	165	390	705
Mesh	102	409	939	1635

In the following simulations, the processing time $P = 16$ and the transfer length $D = 1024$ words which sets tight requirements for the communication network. This kind of communication intensive transfer patterns can be found, for example, in packet processing inside an Internet router. The measured execution times are listed in Table 5. The poor result of the single bus in case 1 is due to inefficiency of the utilized distributed round-robin arbitration. The transfer times of the mesh and the hierarchical bus are quite close to each other due to sequential nature of the application. In test case 2, the differences between the networks are more apparent. The transfer times of the single bus grow again very fast. On the other hand, the results for hierarchical bus and mesh are quite close to each other and follow the results predicted by (1), which means that both networks are able to systematically exploit the inherent parallelism. The same happens with test cases 3 and 4. The run-time of case 5 is defined by the slowest individual application when applications 1-4 are run together. Results of case 5 with 64 agents are not available due to a limitation in the current simulation environment.

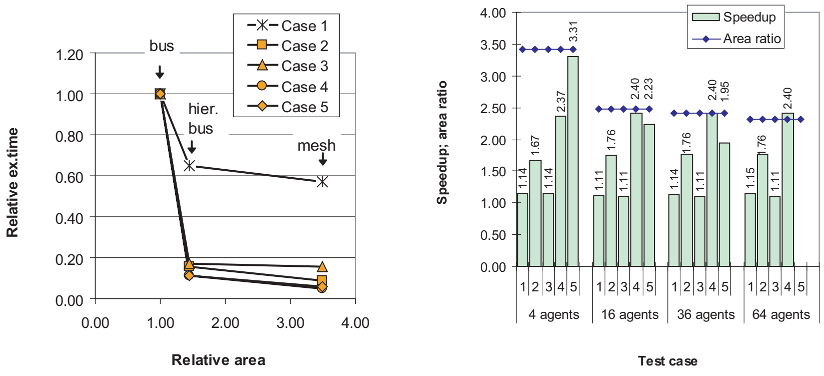
Table 5. The execution times in clock cycles for test cases

N	4 agents					16 agents				
Test case	1	2	3	4	5	1	2	3	4	5
Single bus	17 034	9 788	17 034	9 217	76 278	68 232	38 971	43 008	36 864	933 504
Hier. bus	17 034	9 788	17 034	9 217	76 278	69 325	13 677	16 527	9 346	222 519
Mesh	14 913	5 858	14 913	3 886	23 040	62 240	7 793	14 913	3 886	99 860
N	36 agents					64 agents				
Test case	1	2	3	4	5	1	2	3	4	5
Single bus	245 556	87 535	96 768	82 944	4 171 264	663 616	233 456	172 032	147 456	x
Hier. bus	159 865	13 727	16 527	9 346	465 375	286 620	13 782	16 527	9 346	x
Mesh	140 040	7 808	14 913	3 886	238 878	248 950	7 823	14 913	3 886	x

The results show that the single bus is clearly not applicable for large systems. Mesh is the fastest network in all cases but also the biggest. Fig 3(a) shows the Pareto curves for all test cases with 36 agents. Results are scaled so that both area and execution time of single bus equal one. Fig 3(b) shows the measured speedup of mesh over hierarchical bus. Speedup is defined as hierarchical bus execution time divided by mesh execution time. It also shows the area ratio, which is the area of mesh divided by the area of hierarchical bus. Mesh is faster than hierarchical bus but often the area overhead is bigger than the speedup; especially in cases 1 and 3 that do not offer much parallelism. The choice between the mesh and the hierarchical bus is, therefore, a trade-off between area cost and performance. For comparison, we define the architectural performance as the inverse of the costs:

$$Performance = cost^{-1} = ({t_{\mathrm{tot}}}^{w_t} * A)^{-1} \; . \tag{3}$$

The cost is defined as a product of execution time t_{tot} and area A. The weighting factor w_t can be utilized to make the runtime less ($w_t < 1$) or more important ($w_t > 1$) than the area. In these cases, smaller weights favor hierarcrhical bus and large ones favor mesh. Fig 4 shows both estimated and measured performance of all networks so that the best case is scaled to 1 and execution and area have equal weights ($w_t = 1$).



(a) Pareto curves for 36 agents

(b) Speedup and area overhead of mesh over hierarchical bus

Fig. 3. Relation of execution time and area

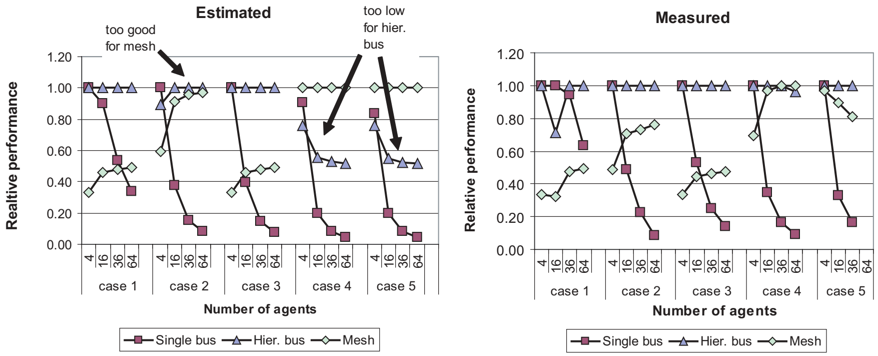


Fig. 4. Relative overall performance

Hierarchical bus offers the best trade-off for test cases 1, 2, 3, and 5. Mesh is best suited for the most parallel test case, that is test case 4 with large number of agents. A single bus is applicable only in sequential test case 1.

The shapes of the estimated and the measured performance curves match rather well. This implies that equation (1) predicts the ratio between execution times well in many cases. However, in some cases, denoted with arrows in Fig 4, the estimates are clearly inaccurate. Furthermore, the estimated times are much smaller than the measured results. This is mainly due to the implementation of TG network contention, and inefficiencies in implemented network protocols that were not included in equations. Furthermore, other than 1-to-1 mappings having more contention are likely to cause even bigger error in estimates.

6 Conclusions

This paper presented a general way for fair comparison of networks for large SoCs using synthesized HW models. Single bus, hierarchical bus, and 2-dimensional mesh networks were used in this study. Application dependence is a factor that cannot be disregarded in communication-based system design. Therefore, the analysis utilizes Transaction Generator that makes it possible to simulate the networks with different traffic patterns. Results show that theoretical performance derived from the number of links in the network does not reflect the application execution time linearly. The presented formal equation provides reasonable estaimtes in some case, but not in general. Moreover, more advanced estimates would require rather complex equations. Therefore, fast, cycle-accurate simulation is preferred.

Many contemporary analyzes often depict buses as poor fits to large systems because only the single bus is used as a reference. The presented hierarchical bus scales quite easily to large systems and provides a good area-performance trade-off while retaining many of the advantageous features of simpler bus arrangements. The hierarchical bus exhibits good run-time results with relatively small implementation area. The analyzed 2-dimensional mesh network provides the highest performance with the largest area. The architectural performance, defined as a product of area and execution time, favors the use of hierarchical bus. There is on-going work for developing more test cases (both synthetic and profiled real applications), exploring different process mappings, and including other network topologies. Furthermore, more performance and cost factors, such as energy and latency variation, will be analyzed. Such metrics are definitely more complex to analyze formally but can be estimated through simulation.

References

1. Benini, L., de Micheli, G.: Networks on chips: a new SoC paradigm. Computer **35** (2002) 70–78
2. Ho, R., Mai, K.W., Horowitz, M.A.: The future of wires. Proc. IEEE **89** (2001) 490–504
3. Sylvester, D., Keutzer, K.: Impact of small process geometries on microarchitectures in systems on a chip. Proc. IEEE **89** (2001) 467–489
4. Salminen, E., Lahtinen, V., Kuusilinna, K., Hämäläinen, T.D.: Overview of bus-based system-on-chip interconnections. In: ISCAS, Scottsdale, Arizona, USA (2002) 372–375
5. Lines, A.: Asynchronous interconnect for synchronous SoC design. Micro **24** (2004) 32–41
6. Liang, J., Laffely, A., Srinivasan, S., Tessier, R.: An architecture and compiler for scalable on-chip communication. TVLSI **12** (2004) 711–726
7. Wiklund, D., Sathe, S., Liu, D.: Network on chip simulations for benchmarking. In: IWSOC, Banff, Canada (2004) 269–274
8. Andriahantenaina, A., Charlery, H., Greiner, A., Mortiez, L., Zeferino, C.A.: SPIN: a scalable, packet switched, on-chip micro-network. In: DATE, Munich, Germany (2003) 70–73
9. Bartic, T.A., Mignolet, J.Y., Nollet, V., Marescaux, T., Verkest, D., Vernalde, S., Lauwereins, R.: Highly scalable network on chip for reconfigurable systems. In: Symposium on System-on-Chip, Tampere, Finland (2003) 79–82
10. Moraes, F., Calazans, N., Mello, A., Möller, L., Ost, L.: HERMES: an infrastructure for low area overhead packet-switched networks on chip. Integration, the VLSI journal **38** (2004) 69–93

11. Saastamoinen, I., Alho, M., Nurmi, J.: Buffer implementation for Proteo network-on-chip. In: ISCAS, Bangkok, Thailand (2003) 113–116
12. Lahiri, K., Raghunathan, A., Dey, S.: Evaluation of the traffic-performance characteristics of system-on-chip communication architectures. In: Conference on VLSI design, Bangalore, India (2001) 29–35
13. Thid, R., Millberg, M., Jantsch, A.: Evaluating NoC communication backbones with simulation. In: Norchip, Riga, Latvia (2003) 27–30
14. Erbas, C., Polstra, S., Pimentel, A.D.: IDF models for trace transformations: A case study in computational refinement. In: SAMOS, Samos, Greece (2003) 167–172
15. Kangas, T., Riihimäki, J., Salminen, E., Kuusilinna, K., Hämäläinen, T.D.: Using a communication generator in SoC architecture exploration. In: Symposium on System-on-Chip, Tampere, Finland (2003) 105–108
16. Kreutz, M.E., Carro, L., Zeferino, C.A., Susin, A.A.: Communication architectures for system-on-chip. In: SBCCI, Pirenopolis, Brazil (2001) 14–19
17. Zeferino, C.A., Kreutz, M.E., Carro, L., Susin, A.A.: A study on communication issues for systems-on-chip. In: SBCCI, Porto Alegre, Brazil (2002) 121–126
18. Zhang, H., Wan, M., George, V., Rabaey, J.: Interconnect architecture exploration for low-energy reconfigurable single-chip DSPs. In: Workshop on VLSI, Orlando, Florida, USA (1999) 2–8
19. Kahn, G.: The semantics of a simple language for parallel programming. In: IFIP Conference, Stockholm, Sweden (1974) 471–475

DDM-CMP: Data-Driven Multithreading on a Chip Multiprocessor

Kyriakos Stavrou, Paraskevas Evripidou, and Pedro Trancoso

Department of Computer Science, University of Cyprus,
75 Kallipoleos Ave., P.O.Box 20537, 1678 Nicosia, Cyprus
{tsik,skevos,pedro}@cs.ucy.ac.cy

Abstract. High-end microprocessors achieve their performance as a result of adding more features and therefore increasing their complexity. In this paper we present DDM-CMP, a Chip-Multiprocessor using the Data-Driven Multithreading execution model.

As a proof-of-concept we present a DDM-CMP configuration with the same hardware budget as a high-end processor. In that budget we implement four simpler CPUs, the TSUs, and the interconnection network. An estimation of DDM-CMP performance for the execution of SPLASH-2 kernels shows that, for the same clock frequency, DDM-CMP achieves a speedup of 2.6 to 7.6 compared to the high-end processor. A lower frequency configuration, which is more power-efficient, still achieves high speedup (1.1 to 3.3). These encouraging results lead us to believe that the proposed architecture has a significant benefit over traditional designs.

1 Introduction

Current state-of-the-art microprocessor designs aim at achieving higher performance by exploiting more ILP through using complex hardware structures. Nevertheless, such increase in complexity results in several problems and consequently marginal performance increases. Palacharla *et al.* [1] explain that window wakeup, selection and operand bypass logic are likely to be the most limiting factors for improving performance in future designs. The analysis presented by Olukotun *et al.* [2] proves that the complexity a large number of structures increases in a quadratic way with different processor parameters such as the issue width and the number of pipeline stages. Increasing design complexity does not only limit the performance improvement but also makes the validation and testing a difficult task [3].

Agarwal *et al.* [4] derive that the doubling of microprocessor performance every 18 months has been the result of two factors: more transistors per chip and superlinear scaling of the processor clock with technology generation. Their results show that, due to both diminishing improvements in clock rate and poor wire scaling as semiconductor devices shrink, the achievable performance growth of conventional microarchitectures will slow down substantially.

An alternative design that achieves parallelism but avoids the complexity is the Chip Multiprocessor (CMP) [2]. Several research projects have proposed CMP architec-

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 364–373, 2005.

tures [2, 5, 6, 7]. In addition, commercial products have also been proposed (e.g. IBM's Power5 [8] and SUN's Niagara [9]).

Parallel architectures often suffer from large synchronization and communication latencies. Data-Driven Multithreading (DDM) [10, 11] is an execution model that aims at tolerating the latencies by allowing the computation processor to produce useful work while a long latency event is in progress. In this model, the synchronization part of the program is separated from the communication part allowing it to hide the synchronization and communication delays [10]. While such computation models usually require the design of dedicated microprocessors, Kyriacou *et al.* [10] showed that the DDM benefits may be achieved using commodity microprocessors. The only additional requirement is a small hardware structure, the Thread Synchronization Unit (TSU).

The contribution of this paper is to explore the DDM concept with the new CMP type of architectures. The proposed architecture, DDM-CMP, is a chip multiprocessor architecture where the cores are simple embedded processors operating under the DDM execution model. Along with the cores, the chip also includes the TSUs and an interconnection network. The use of embedded processors is justified by Olukotun *et al.* [2] who showed that the simpler the cores of the multiprocessor, the higher their frequency can be. In addition, embedded processors are smaller and therefore we are able to include more cores in the same chip. A prototype will be build using Xilinx Virtex II Pro [12] chip.

Our experiments use kernels from SPLASH-2 benchmark suite and compare the estimated performance of a DDM-CMP system composed of four simple cores to that of an equal hardware budget high-end processor. For this analysis we use Pentium III and Pentium 4 as representatives of simple and high end processors, respectively. The results show that a DDM-CMP configuration clocked at the same frequency with the high-end processor achieves a speedup of 2.6 to 7.6. DDM-CMP's benefits may be explored for low-power configurations as the results show that even when clocked at a frequency less than half of the high-end processor's, it achieves a speedup of 1.1 to 3.3.

The rest of this paper is organized as follows. Section 2 describes DDM execution model, the proposed DDM-CMP architecture and its prototype implementation. Section 3 describes the case study used as the proof of our concept. Finally we present our conclusions in Section 4.

2 DDM-CMP Architecture

The proposed DDM-CMP architecture is the evolution of the DDM architecture presented in [10, 11]. In this section, we present the DDM model of execution and describe the DDM-CMP architecture, its prototype implementation and its target applications.

2.1 DDM Model

Data-Driven Multithreading (DDM) provides effective latency tolerance by allowing the computation processor produce useful work, while a long latency event is in progress. This model of execution has been evolved from the dataflow model of computation. In particular, it originates from the dynamic dataflow Decoupled Data Driven (D^3)

graphs [13, 14], where the synchronization part of a program is separated from the computation part. The computation part represents the actual instructions of the program executed by the computation processor whereas the synchronization part contains information about data dependencies among threads and is used for thread scheduling.

A program in DDM is a collection of re-entrant code blocks. A code block is equivalent to a function or a loop body in the high-level program text. Each code block comprises of several threads. A thread is a sequence of instructions equivalent to a basic block. A producer-consumer relationship exists among threads. In a typical program, a set of threads called the producers create data used by other threads called the consumers. Scheduling of code blocks, as well as scheduling of threads within a code block, is done dynamically at run time according to data availability. While the instructions within a thread are fetched by the CPU sequentially in control-flow order, the CPU may apply any optimization to increase ILP (e.g. out-of-order execution).

As we are still in the process of developing a DDM-CMP compiler, the procedure of partitioning the program into a data-driven synchronization graph and code threads (as presented in [15]) is currently done by hand.

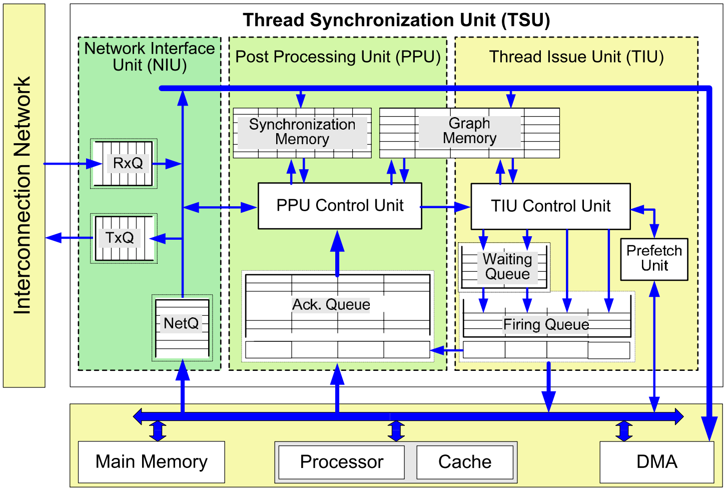


Fig. 1. Thread Scheduling Unit (TSU) internal structure

TSU - Hardware Support for DDM. The purpose of the Thread Synchronization Unit (TSU) is to provide hardware support for data-driven thread synchronization on conventional microprocessors. The TSU is made out of three units: The Thread Issue Unit (TIU), the Post Processing Unit (PPU) and the Network Interface Unit (NIU). When a thread completes its execution, the PPU updates the Ready Count (Ready Count is set by the compiler and corresponds to the number of input values or producers to the thread) of its consumer threads, determines whether any of those threads became ready for execution and if so, it forwards them to the TIU. The function of the TIU is to schedule and prefetch threads deemed executable by the PPU. The NIU is responsible for the communication between the TSU and the interconnection network. The internal structure of the TSU is depicted in Figure 1. A detailed description of the operation of the TSU can be found in [15].

CacheFlow. Although DDM can tolerate communication and synchronization latency, scheduling based on data availability may have a negative effect on locality. To overcome this problem, the scheduling information together with software-triggered data prefetching, are used to implement efficient cache management policies. These policies are named CacheFlow [16]. The most effective CacheFlow policy contains two optimizations *False Conflict Avoidance* and *Thread Reordering*.

False conflict avoidance prevents the prefetcher from replacing cache blocks required by the threads deemed executable, and so reduces cache misses. Thread reordering attempts to exploit both temporal and spatial locality by reordering the threads still waiting for their input data.

2.2 CMP Architecture

The proposed chip multiprocessor can be implemented using three hardware structures: the *microprocessor cores*, the *TSUs* and the *interconnection network*.

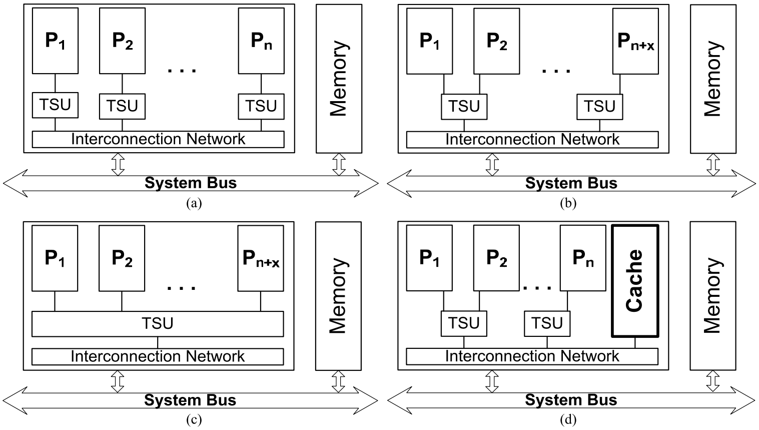


Fig. 2. Several alternatives of the DDM-CMP architecture: (a) Each microprocessor has its own TSU, (b) One TSU is shared among two microprocessors and the number of cores increases, (c) One TSU serves all the microprocessors of the chip, and (d) Saved space is used to implement on-chip shared cache

Our first proposed CMP architecture (Figure 2-(a)) is one that simply performs the integration of the previously proposed D^{2}NOW [11] into a single chip. While having one TSU per processor was required in a NOW system, when all processors are on the same chip it is possible to optimize the use of the TSU structure and share it among two or more processors (Figure 2-(b)). Ultimately we may consider the extreme case where one TSU is shared among all CPUs on-chip (Figure 2-(c)). Notice that by saving hardware with the sharing of the TSUs it may be possible to increase the number of on-chip CPUs or alternatively add internal shared cache (Figure 2-(d)).

Although the impact of the interconnection network to the performance of an architecture that uses DDM execution model is small [15], there is still potential for studying several alternatives. This is specially interesting as the number of on-chip CPUs increases. The tradeoff between the size and the performance of the interconnection network will be studied as a larger, more complex, interconnection network may result in a decrease of the number of CPUs that can be embedded in the chip.

2.3 Prototype Implementation

To prove that the proposed DDM-CMP architecture offers the expected benefits, a hardware prototype will be implemented. This prototype will use the Xilinx Virtex II Pro chip [12]. This chip contains, among others, two embedded Power PC 405 [17] processors and a programmable FPGA with more than 30000 logic cells. We aim at implementing the TSU and the interconnection network on the FPGA portion and execute the application threads on the two processors.

2.4 Target Applications

The DDM-CMP architecture can be used to speedup the execution of parallelizable loop-based or pipeline-like applications. On the one hand, the proposed architecture is explicitly beneficial for parallelizable applications as it provides multiple parallel execution processors. On the other hand, protocol stack applications, that are representative examples of pipeline-like applications, can benefit from DDM-CMP, by mapping the code corresponding to each layer to a different DDM thread. Each layer, or DDM-thread, will be running in parallel providing a pipelined execution model with significant performance enhancement.

Overall, we envision the DDM-CMP chip to be used in a single chip system as a substitute of a high-end microprocessor or as a building block for larger multiprocessor systems like BlueGene/L [18].

3 DDM-CMP Performance Potential Analysis

3.1 Design

The objective of the proposed DDM-CMP architecture is to achieve better performance than a current high-end microprocessor, given the same hardware budget, *i.e.* the same die area. For our analysis we consider the Intel Pentium 4 as the baseline for the high-end microprocessor. As mentioned before, DDM-CMP is build out of simpler cores. For the purposes of our analysis we consider Intel Pentium III as a representative of such a core. From the information reported in [19], the number of transistors used in implementing Intel Pentium 4 3.2GHz 1MB L2 cache 90nm technology is approximately

125 million while the number of transistor used in implementing the Intel Pentium III 800MHz 256KB L2 cache 180nm technology is 22 million. Therefore, the Pentium 4 requires approximately 5.7 times more transistors than what is needed to build Pentium III.

In addition to the processors, other hardware structures are needed to implement the DDM-CMP architecture: the TSUs and the interconnection network. As explained earlier, these structures can be implemented using a relatively small number of transistors. If we use the Pentium 4 chip in order to implement four Pentium III processors, about 37 million transistors will be left unused. This number of transistors is more that enough to implement the four TSUs and the appropriate interconnection network. Therefore, a DDM-CMP architecture with four Pentium III processors can be implemented with the same number of transistors needed to build a Pentium 4. This is the DDM-CMP configuration that will be used for our proof-of-concept experiments.

3.2 Experimental Setup

As we do not have yet a DDM-CMP simulator, its performance results are derived from the results obtained by Kyriacou *et al.* [10] for the D^2NOW implementation. In this case, we will use the results for the D^2NOW architecture configured with four Pentium III 800MHz processors including all architecture optimizations. Notice that the D^2NOW results are conservative for the DDM-CMP architecture as the on-chip interconnection network has both larger bandwidth and smaller latency than the D^2NOW interconnect.

As for the baseline high-end processor we have selected the Pentium 4 3.2GHz. To obtain the results for this setup we measure the execution time of the application's native execution on that system. The execution time is determined by measuring the number of processor cycles consumed in the execution of the main function of the program, *i.e.* we ignore the initialization phase. The processor cycles are measured by reading the contents of the hardware program counter of the processor [20]. Notice that as this is native execution, in order for the results to be statistically significant we execute the same experiment ten times and exclude the largest and smaller measurement.

For this proof-of-concept analysis the workload considered to test the proposed architecture is composed of three kernels from the SPLASH-2 benchmark suite [21]: LU, FFT, and Radix.

3.3 Experimental Results

The results collected from [10] and from the native execution on the Pentium 4 system are summarized in Table 1. It is possible to observe that the 4 x Pentium III DDM-CMP achieves better performance than the native Pentium 4 for only the Radix application. For both FFT and LU the DDM-CMP performance is worse than the one obtained for the Pentium 4. It is interesting to note that these results may be correlated with the fact that for both FFT and LU the execution time is much smaller than the one for the Radix application. From a brief analysis of the execution of the applications we were able to determine that the main function that performs the calculations for the algorithm accounts for more than 80% of the total execution for Radix, while it accounts for approximately only 50% for both FFT and LU. This is an indication that in order to

Table 1. Performance results with different implementation technology

	DDM-CMP **4 x Pentium III 800MHz**		**CPU** **Pentium 4 3.2GHz**		**Speedup**
	Cycles [x1000]	**Time [ms]**	**Cycles [x1000]**	**Time [ms]**	
FFT	29825	37.3	80283	25.1	0.67
LU	25319	31.6	66485	20.8	0.66
Radix	125491	156.9	952970	297.8	1.90

obtain more reliable results for both FFT and LU we will need to use larger data set sizes. This issue will be covered in the near future as we complete the DDM-CMP simulator. Nevertheless, at this point we do not consider this result to be a problem as we expect that there will always be applications that will not show better performance when executing on the DDM-CMP.

The results presented in Table 1 are significantly affected by technology scaling. It is important to notice that the Pentium III is implemented using 0.18μm technology and its clock speed is 800MHz whereas the Pentium 4 is implemented using 0.09μm technology and a clock of 3.2GHz. If pipeline stalls due to off-chip operations are not taken into account, the number of clock cycles needed to execute a series of instructions is independent of the implementation technology. If we consider that the off-chip operations are not dominant in the applications studied, the execution time for the specific application on the specific architecture will decrease with the same rate that the frequency increases. In this analysis we will consider two frequency scaling scenarios. The first one is a realistic scaling where instead of considering the original Pentium III 800MHz we consider the highest clock frequency which it was produced. From [22], the Pentium III code named Tualatin had a clock frequency of 1.4GHz. The second scaling is the upper most limit scenario where we consider that we would use a Pentium III equivalent processor which would be able to scale up to the Pentium 4 frequency (3.2GHz).

An additional optimization that will be considered in the future is the fact that the TSU may be modified to be shared among more than one processor. This will minimize the hardware overhead that is introduced due to the DDM architecture. The extra space that is saved from sharing the TSU may be used to increase the number of cores on the chip.

One more factor that will have an impact on the performance of DDM-CMP is the type of processor used as the core. In this analysis we are using the Pentium III given the restrictions that originate from the use of the results obtained with the D^{2}NOW study. In a real implementation, as we discussed previously, we will use embedded processors as the core for the DDM-CMP. As these embedded processors are simpler, they require fewer transistors for their implementation and consequently we will be able to fit more cores into the same chip. Given the above arguments, in addition to the frequency scaling, we also consider the case where we would be able to fit eight processors on the same chip. Notice that as we use the results from a D^{2}NOW configured with eight Pentium III processors the results are not accurate, but they can be used as an indication of the upper limit that may be achieved. The results from these scaling scenarios together with the original results are depicted in Figure 3.

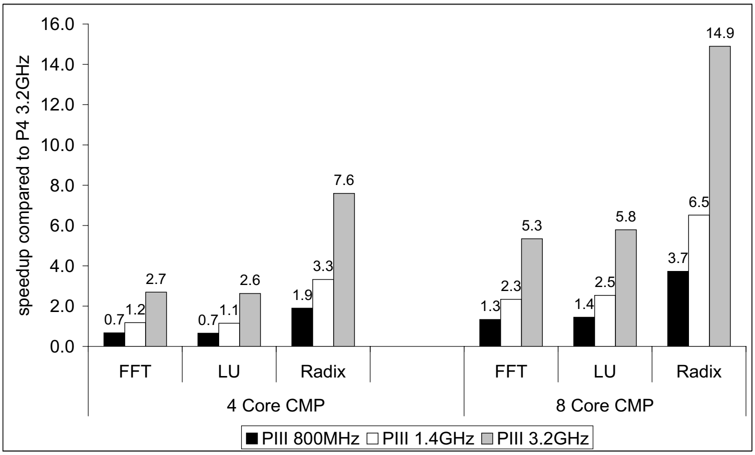


Fig. 3. Speedup when frequency and core scaling is taken into account

In Figure 3 we have depicted three bars for each application. The one on the left represents the original results, the one in the middle represents the Pentium III scaling for 1.4GHz, and the one on the right represents the upper limit scaling with the 3.2GHz clock frequency. The group of results on the left represent the original chip design with four cores while the group on the right represents the scenario where the optimizations used allowed for the scaling of the number of cores to eight.

As it was already observed with the original results, both FFT and LU have a speedup smaller than 1 and therefore their performance is better on the Pentium 4 system. In contrast, Radix achieves almost a 2x speedup when executing on the original DDM-CMP. The speedup values increase as the scaling is applied to the Pentium III processor. It is relevant to notice that even with the first scaling all three applications already show a better performance on the DDM-CMP compared to the execution on the Pentium 4. This configuration has also the advantage of being more power-efficient than the original Pentium 4 as it is clocked at less than half of its frequency. When scaling the frequency to the same as the baseline we observe larger speedup values ranging from 2.6 to 7.6. It is also interesting to observe that Radix presents a superlinear speedup as with the upper limit scaling it achieves a speedup of 7.6 with only four processors. This may be justified by the effectiveness of the CacheFlow policies.

The results for the eight core DDM-CMP present, for all applications at any scaling scenario, a speedup larger than one.

Overall, the results show very good speedup for both the high-performance and low-power DDM-CMP configurations.

4 Conclusions

In this paper we have presented DDM-CMP, a Chip-Multiprocessor implementation using the Data-Driven Multithreading execution model. The DDM-CMP architecture turns away from the complexity path taken by recent high-end microprocessors. Its performance is achieved by combining several simple commodity microprocessors together with a small overhead, an extra hardware structure, the Thread Scheduling Unit (TSU).

As a proof-of-concept we present a DDM-CMP implementation that utilizes the same hardware budget as a current high-end processor, the Pentium 4, to implement four Pentium III processors together with the necessary TSUs and interconnection network. The results obtained are very encouraging as the DDM-CMP configuration clocked at the same frequency as the Pentium 4 achieves a speedup of 2.6 to 7.6. DDM-CMP can alternatively be configured for power-efficiency and still achieve high speedup. A configuration clocked at less than half of the Pentium 4 frequency achieves speedup values ranging from 1.1 to 3.3. We are currently evaluating the different architecture alternatives for DDM-CMP, a larger set of applications, and are starting to implement a prototype of this architecture on a Virtex II Pro chip.

Acknowledgments

We would like to thank Costas Kyriacou for his contribution in the discussions and preparation of the results. Also, we would like to thank the anonymous reviewers for their valuable comments.

References

1. Palacharla, S., Jouppi, N., Smith, J.: Complexity Effective Superscalar Processors. In: Proc. of the 24th ISCA. (1997) 206–218
2. Olukotun, K., *et al.*: The Case for a Single Chip Multiprocessor. In: Proc. of the 7th ASPLOS. (1996) 2–11
3. Silas, I., *et al.*: System-Level Validation of the Intel(r) Pentium(r) M Processor. Intel Technology Journal **7** (2003)
4. Agarwal, V., *et al.*: Clock rate versus IPC: The end of the Road for Conventional Microarchitectures. In: Proc. of the 27th ISCA. (2000) 248–259
5. Hammond, L., *et al.*: The Stanford Hydra CMP. IEEE Micro **20** (2000) 71–84
6. Barroso, L., *et al.*: Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In: Proc. of the 27th ISCA. (2000) 282–293
7. Taylor, M., *et al.*: Evaluation of the Raw Microprocessor: An Exposed Wire Delay Architecture for ILP and Streams. In: Proc. of the 31st ISCA. (2004) 2–13
8. Kalla, R., Sinharoy, B., Tendler, M.: IBM POWER5 Chip: A Dual-Core Multithreaded Processor. IEEE Micro **24** (2004) 40–47
9. Kongetira, P.: A 32-way Multithreaded SPARC Processor. In: Proc. of Hot Chips 2004. (2004)
10. Kyriacou, C., Evripidou, P., Trancoso, P.: Data Driven Multithreading Using Conventional Microprocessors. Technical Report TR-05-4, University of Cyprus (2005)
11. Evripidou, P., Kyriacou, C.: Data driven network of workstations (D2NOW). J. UCS **6** (2000) 1015–1033
12. XILINX: Virtex-II Pro and Virtex-II Pro X FPGA User Guide. Version 3.0 (2004)

13. Evripidou, P.: D3-machine: A Decoupled Data Driven Multithreaded architecture with variable resolution support. Parallel Computing **27** (2001) 1015–1033
14. Evripidou, P., Gaudiot, J.: A decoupled graph/computation data-driven architecture with variable resolution actors. In: Proc. of ICPP 1990. (1990) 405–414
15. Kyriacou, C.: Data Driven Multithreading using Conventional Control Flow Microprocessors. PhD dissertation, University of Cyprus (2005)
16. Kyriacou, C., Evripidou, P., Trancoso, P.: CacheFlow: A Short-Term Optimal Cache Management Policy for Data Driven Multithreading. In: Proc. of the 10th Euro-Par, Pisa, Italy. (2004)
17. IBM Microelectronics Division: The PowerPC 405(tm) Core (1998)
18. The BlueGene/L Team: An Overview of the BlueGene/L Supercomputer. In: Proc. of the 2002 ACM/IEEE supercomputing. (2002) 1–28
19. Intel: Intel Microprocessor Quick Reference Guide. http://www.intel.com/ pressroom/kits/quickreffam.htm (2004)
20. PCL: The Performance Counter Library Version 2.2 (2003)
21. Woo, S., *at.*: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: Proc. of 22nd ISCA. (1995) 24–36
22. Topelt, B., Schuhmann, D., Volkel, F.: The mother of all CPU charts Part 2. http://www6.tomshardware.com/cpu/20041221/index.html (2004)

Modeling NoC Architectures by Means of Deterministic and Stochastic Petri Nets

H. Blume, T. von Sydow, D. Becker, and T.G. Noll

Chair for Electrical Engineering and Computer Systems,
RWTH Aachen University, Schinkelstraße 2, 52062 Aachen
{blume, sydow, dbecker, tgn}@eecs.rwth-aachen.de

Abstract. The design of appropriate communication architectures for complex Systems-on-Chip (SoC) is a challenging task. One promising alternative to solve these problems are Networks-on-Chip (NoCs). Recently, the application of deterministic and stochastic Petri-Nets (DSPNs) to model on-chip communication has been proven to be an attractive method to evaluate and explore different communication aspects. In this contribution the modeling of basic NoC communication scenarios featuring different processor cores, network topologies and communication schemes is presented. In order to provide a test bed for the verification of modeling results a state-of-the-art FPGA-platform has been utilized. This platform allows to instantiate a soft-core processor network which can be adapted in terms of communication network topologies and communication schemes. It will be shown that DSPN modeling yields good prediction results at low modeling effort. Different DSPN modeling aspects in terms of accuracy and computational effort are discussed.

1 Introduction

With the advent of heterogeneous Systems-on-Chip (SoCs), on-chip communication issues are becoming more and more important. As the complexity of SoCs is increasing a variety of appropriate communication architectures are discussed, ranging from basic bus oriented to highly complex packet oriented Network-on-Chip (NoC) structures [1], [2]. These communication structures have to be evaluated and quantitatively optimized presumably in an early stage of the design process. Various approaches have been developed in order to evaluate SoC communication performance and to compare architectural alternatives. Examples for such communication modeling approaches are:

- simulative approaches, e. g. applying SystemC [3], [4],
- combined simulative-analytic approaches [5],
- formal communication modeling and refinement systems applying dedicated modeling languages like the Action Systems Formalism [6],
- stochastic approaches applying Markov Models [7], Queuing Theory [8] or deterministic and stochastic Petri Nets [9], [10].

Each of these techniques provides its individual advantages and disadvantages. For example, simulative approaches like [3] provide highly accurate results but suffer from

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 374–383, 2005.

long simulation times, making them not appropriate for an early stage of communication modeling and evaluation. Recently, communication modeling approaches which are based on so-called deterministic and stochastic Petri-Nets (DSPNs) have been presented. In [9], it could be shown that applying these DSPN modeling techniques it is possible to efficiently trade modeling effort and modeling accuracy. Applying very simple but exemplary test scenarios like resource conflicts in state-of-the-art DSP architectures and basic bus-based communication test cases a very good modeling accuracy with low modeling effort could be achieved. In order to prove that these techniques are also suitable for more complex communication scenarios the application of these DSPN-based modeling techniques to NoC-communication problems like on-chip multi-processor communication is investigated in this paper. In order to verify the modeling results a generic NoC test bed has been built first. Therefore, an FPGA-based platform has been utilized on which several proprietary so-called soft-core processors (Nios [11]) besides other components like DMA-controllers, on-chip memories or dedicated (custom-made) logic blocks can be instantiated and connected using different communication architectures. The FPGA based generic platform allows to determine NoC performance in terms of latency, throughput respectively bandwidth etc. These results will be compared to the results which were achieved by the DSPN model. Due to limited hardware resources, highly complex communication scenarios with for example different topologies of processor clusters (different hierarchy levels etc.) are not emulated using RISC-like Nios soft-cores. Instead, such processor networks were tested with rudimentary processor cores which implement the functionality of a sender and/or a receiver. The following issues have been addressed within this contribution: Modeling effort, modeling accuracy and required computation time depending on the DSPN solving methods. The paper is organized as follows: Chapter 2 sketches the basics of DSPN modeling. In chapter 3 some details on the FPGA-based test bed which has been utilized in the experiments are described. The modeling of NoC test scenarios and the corresponding results are described in chapter 4. Conclusions are given in chapter 5.

2 Short Synopsis of Modeling with DSPNs

A comprehensive overview of the modeling possibilities with deterministic and stochastic Petri nets (DSPNs) and all corresponding options are not in the scope of this paper. Here, only those basics will be shortly sketched which are used in the following sections. DSPNs consist of so-called places, arcs and transitions. Places, depicted as circles in the graphical representation, are states of system components. E.g. a place could be named *compute result* to make clear that this place represents the state of computing a result in the belonging component. Places can be untagged or marked with one or more tokens which illustrate that the corresponding place is actually allocated. The change of a state can be described by so-called transitions. Three types are differentiated: There are immediate transitions, transitions with a probability density function for the delay (e.g. negative exponential) or deterministically delayed transitions. Furthermore, priorities can be assigned to each transition. Transitions and places are connected via arcs. Arcs can be of two types, regular or inhibitor. Inhibitor arcs are identified by a small inversion circle instead of an arrowhead at the destination end of it (see Fig. 1). If more

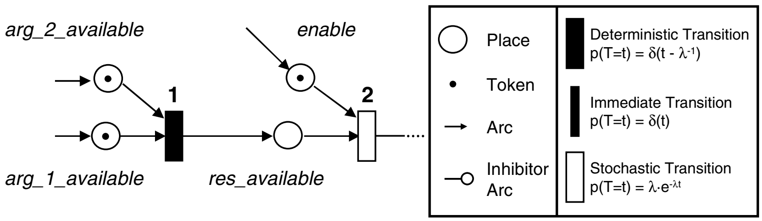


Fig. 1. Exemplary Section out of a deterministic and stochastic Petri-net model

than one input place is connected to a transition via regular arcs, the transition will only be activated when all connected places are marked. If one or more of these arcs is an inhibitor arc the transition will not fire if the corresponding place is marked. Once a Petri net model is implemented, performance measures, such as marking probabilities and the expected number of tokens for places and throughput for deterministic and exponential transitions can be defined and subsequently computed by simulation, mathematical approximation or mathematical analysis.

The different alternatives vary in terms of accuracy and computational effort. E.g. in the case of two or more concurrently enabled deterministic transitions, mathematical analysis is not possible [12]. Fig. 1 depicts a section of a very simple modeling example where the transition 1 will fire only when both connected places (*arg_1_available*, *arg_2_available*) contain at least one token. Then this transition will fire with a deterministic delay time, modeling the processing time for e.g. a computational kernel. The corresponding delay time T1 for this transition is a configuration parameter of this DSPN. After this delay time has been elapsed, one token will be taken from all of the transitions input places (here: *arg_1_available* and *arg_2_available*), and one token will be placed in all connected output places (here only one: *res_available*). Depending on the status of its other input places (here: *enable*) - the next transition 2 (in this case a stochastic transition) is going to fire with a random delay with an exponential distribution. In principle, each communication scenario can be modeled by DSPNs. Some SoC modeling examples can be found in [9] and a comprehensive overview of modeling with DSPNs is given in [12]. A variety of DSPN modeling environments is available today [13]. In the course of the modeling experiments described here, the DSPN modeling environment DSPNexpress [14] has been used. DSPNexpress provides a graphical editor for DSPN models, as well as a solver backend for numerical analysis of DSPNs. Experiments can be performed for a fixed parameter set and for a parameter sweep across a user-defined range of values. The package supports the computation of the transient response e.g. the distribution of tokens after a certain amount of cycles (using Picards Iteration Algorithm) as well as computation of the steady state behavior of the realized DSPN model. The latter can be realized by iteratively using the Generalized Minimal Residual Method, by employing the direct quadrature method or by utilizing the discrete event simulator backend [12]. These methods correspond to the DSPN computation methods mentioned in the beginning of this chapter.

3 FPGA Based NoC Testbed

As a test bed for the evaluation of NoC communication scenarios and architectures an FPGA-based system has been applied. As a reference platform an FPGA development board featuring an APEX20K200EFC484 FPGA, 1 MByte of flash memory, 256 KBytes of SRAM, serial interfaces (e.g. for downloading the program to the on-board flash memory) and several display options (e. g. LC display) has been used. Multi processor networks have been implemented on this platform by instantiating Nios soft-core processors. The Nios embedded processor is a general-purpose load/store RISC CPU, that can be combined with a number of peripherals, custom instructions, and hardware acceleration units to create a custom system-on-a-programmable-chip solution. The processor can be configured to provide either 16 or 32 bit wide registers and data paths to match given application requirements. Both versions use 16 bit wide instruction words. Version 3.2 of the Nios core, as used here, typically features about 1100 logic elements (LEs) in 16 bit mode and up to 1700 LEs in 32 bit mode incl. hardware accelerators like hardware multipliers. More detailed descriptions of the components can be found in [11]. Based on such a Nios core a processor network consisting of a general communication structure that interfaces various peripherals and devices to various Nios cores can be constructed. The so-called Avalon [15] structure is used to connect devices to the Nios cores. It is a dynamic sizing communication structure that allows devices with different data widths to be connected, with a minimal amount of interfacing logic. In order to realize processor networks on this platform the so-called SOPC (system on a programmable chip) Builder [16] has been applied. It is a tool for composing heterogeneous architectures including the communication structure out of library components such as CPUs, memory interfaces, peripherals and user-defined blocks of logic. The SOPC Builder generates a single system module that instantiates a list of user-specified components and interfaces incl. an automatically generated interconnect logic. It allows modifying the design components, adding custom instructions and peripherals to the Nios embedded processor and configuring the connection network.

4 Modeling NoC Test Scenarios

4.1 Nios-Based NoC Test Scenario

The first analyzed system is composed of two Nios soft-cores which compete for access to an external shared memory (SRAM) interface. Each core is also connected to a private memory region containing the program code and to a serial interface which is used to ensure communication with the host PC. The proprietary communication structure used to interconnect all components of a Nios-based system is called Avalon [15] which is based on a flexible crossbar architecture. The block diagram of this fundamental resource sharing experiment is depicted in Fig. 2. Whenever multiple masters can access a slave resource, SOPC Builder [16] automatically inserts the required arbitration logic. In each cycle when contention for a particular slave occurs, access is granted to one of the competing masters according to a Round Robin arbitration scheme. For each slave, a so-called share is assigned to all competing masters. This share represents the fraction of contention cycles in which access is granted to this corresponding master. Masters

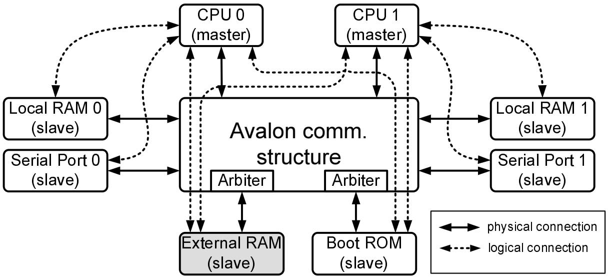


Fig. 2. Block diagram of fundamental resource sharing experiment

incur no arbitration delay for uncontested or acquired cycles. Any masters that were denied access to the slave automatically retry during the next cycle, possibly leading to subsequent contention cycles.

In the modeled scenario the common slave resource for which contention occurs is a shared external memory unit (shaded in gray in Fig. 2) containing data to be processed by the CPUs. Within the scope of this fundamental resource sharing scenario several experiments with different parameter setups have been performed to prove the validity of the DSPN modeling approach. Adjustable parameters include e.g. the priority shares assigned to each processor, the ratio of write and read accesses, the mean delay between memory accesses etc. These parameters have been used to model typical communication requirements of basic operators like digital filters or block read and write operations running on these processor cores. In addition, an experiment simulating a more generic, stochastic load pattern, with exponentially distributed times between two attempts of a processor to access the memory has been performed. Here, each memory access is randomly chosen to be either a read or a write operation according to user-defined probabilities. The distinction between load and store operations is important here because the memory interface can only sustain one write access every two cycles, whereas no such limitation exists for read accesses. The various load profiles were implemented in C, compiled on the host PC and the resulting object code has been transferred to the Nios cores via the serial interface for execution. In the case of the generic load scenario, the random values for the stochastic load patterns were generated in a MATLAB routine. The determined parameters have been used to generate C code sequences corresponding to this load profile. The time between two attempts of a processor to access the memory has been realized by inserting explicit NOPs into the code via inline assembly instructions. Performance measurements for all scenarios have been achieved by using a custom cycle-counter instruction added to the instruction set of the Nios cores. In a first step, a *simple* DSPN model has been implemented (see Fig. 3) in less than two hours. Distinction between read and write accesses was explicitly neglected to achieve a minimum modeling complexity. The DSPN consists of four sub-structures; two parts represent the load generated by the Nios cores (**CPU #1** and **#2**), a simple **cycle process subnet** providing a clock signal and the most complex part being the **arbitration subnet**. Altogether this simple model includes 19 places and 20 transitions. The immediate transitions T1, T2 and T3 and the belonging places P1,

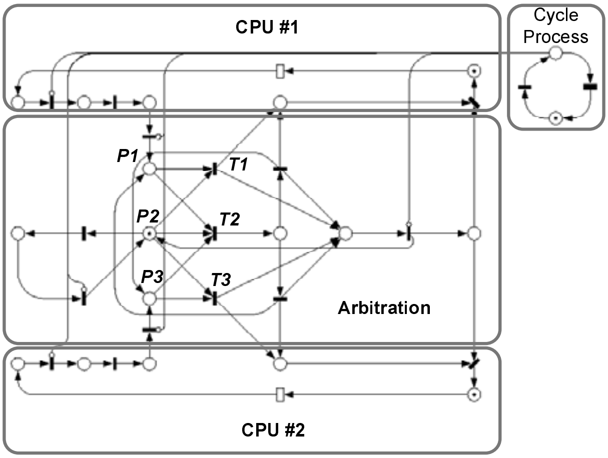


Fig. 3. DSPN for rudimentary Nios example (cycle generator)

P2 and P3 (see Fig. 3) are an essential part of the Round Robin arbitration mechanism implemented in this DSPN. The marked transition P2 denotes that the memory is ready and memory access is possible. P1 and P3 belong to the CPU load processes and indicate that the corresponding CPU (#1, #2) tries to access the memory. If P1 and P2 or P3 and P2 are tagged the transition T1 or accordingly transition T3 will fire and remove the tokens from the connected places (P1, P2 or P2, P3). CPU #1 or CPU #2 has been assigned the memory access in this cycle. A collision occurs if P1, P2 and P3 are tagged with a token. Both CPUs try to access the memory in the same cycle (P1 and P3 marked). Furthermore, the memory is ready to be accessed (P2 marked). A higher priority has been assigned to transition T2 during the design process. This means that if the conditions for all places are equal the transition with the highest priority will fire first. Therefore, T2 will fire and remove the tokens from the places. Thus, the transitions T1, T2 and T3 and the places P1, P2 and P3 handle the occurrence of a collision.

Though the modeling results applying this simple DSPN model are quite accurate (relative error less than 10%, see Fig. 4), it is possible to increase the accuracy even more by extending the modeling effort for the arbitration subnet. For example it is possible to design a DSPN model of the arbitration subnet which properly reflects the differences between read and write cycles. Thus, the arbitration of write and read accesses has been modeled in different processes resulting in different DSPN subnets. This results in a second and more complex DSPN model. The implementation of this *complex* model has taken about three times the effort in terms of implementation time (approximately five hours) than the simpler model described before. Now, the DSPN model consists of 48 transitions and 45 places. Compared to the simple model the maximum error has been further reduced (see Fig. 4). The complex model also properly captures border cases caused e. g. by block read and write operations. The throughput measured for a code sequence containing 200 memory access instructions has been compared to the results of the simple and complex DSPN model. Fig. 4 shows the relative error for

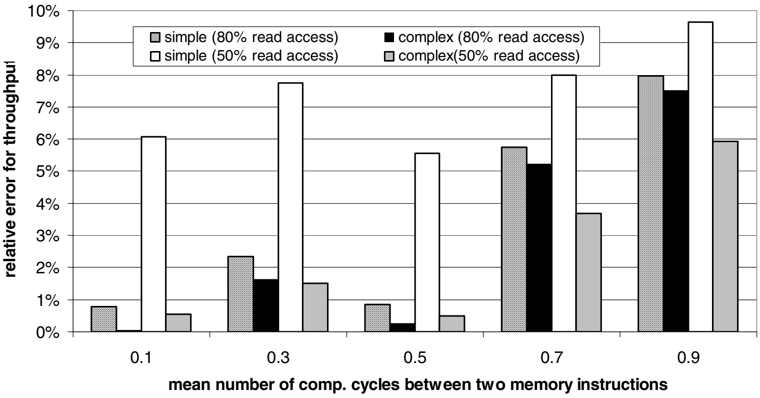


Fig. 4. Comparison of simple and complex DSPN model for different load scenarios

the throughput which is achieved by varying the mean number of computation cycles between two attempts of a processor to access the memory. Furthermore, the results are depicted for two cases, where 50% and 80% of the memory accesses are read accesses, respectively. For example in case of 50% read accesses and a mean of 0.9 computation cycles between two memory accesses, the maximum relative error is reduced from about 10% achieved with the simple model to about 6% with the complex model. As mentioned before, the evaluation of DSPNs can be performed by different methods. The effort in terms of computation time has been compared for a couple of experiments. Generally, the time consumed when applying the simulation method is about two orders of magnitude longer than the time consumed by the analysis methods (direct and iterative). For the example of the complex model the computation time of the DSPN analysis method only amounts to 0.3 sec. and the DSPN simulation time (10^7 memory accesses) amounts to 30 sec. on a Linux-based PC (2.4 GHz, 1 GByte of RAM).

4.2 Generic Processor Core Network

As an example for a more complex network, the DSPN model for a hierarchical network featuring three processor cores, three memory sections and two hierarchically connected switches has been developed. All components are connected by full-duplex links capable of transmitting one data word each clock cycle (see Fig. 5). Because of the limited amount of available logic elements, this network cannot be implemented using Nios cores. Furthermore, SOPC Builder only provides limited configuration options with regard to the Avalon structure in the form of priority shares, and specifically cannot accommodate hierarchical interconnects. Therefore, a flexible generic NoC testbed has been implemented to facilitate implementation of complex networks and arbitrary interconnect structures. To keep the logic element count at a tractable level, the network components are reduced to provide only the functionality that directly affects network traffic. For example, simplified load generators acting as pseudo-CPUs are used instead of Nios cores. Implementation of different routing and arbitration schemes is possible by realizing corresponding interconnection components, e.g. switches. In the example

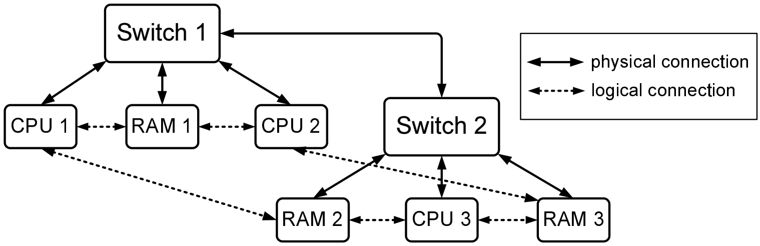


Fig. 5. Exemplary NoC architecture

discussed here, write accesses consist of a single request going from the CPU to the target memory section, whereas read accesses consist of two transactions: An initial request by the CPU, followed by a response from the memory section. The entities for which arbitration has to take place are the output ports of the switches. Access is granted according to a Round Robin scheme with equal priorities. Upon contention, all masters that have been denied access wait until the port is available again. A single-cycle access delay is incurred for the winning master or in the absence of contention. Initiating masters successively acquire all output ports on the path to the addressed slave, and release them once the whole transaction is completed. For the modeled experiment, the processors generate a memory access pattern that is stochastic in space and time: The addressed memory section for each access is determined based on a Bernoulli trial, and the time between consecutive memory accesses for each CPU is exponentially distributed. CPU 1 and 2 generate both local and remote memory accesses, whereas CPU 3 only accesses local memory sections. Fig. 5 sketches the network topology and the logical connections between components.

The DSPN developed to model this setup consists of 64 places and 61 transitions, and provides several parameters defining the network behavior that have been used to set up various experiments. Fig. 6 shows results from two examples: In the experiment whose results are depicted in Fig. 6 a), the ratio between local and remote memory accesses of CPU 1 was varied, and the resulting throughput for all three CPUs was measured. The results show that the mean throughput for CPU 1 is reduced with increasing percentage of remote accesses. This is obvious from the fact that remote transactions require acquisition of two switch ports, whereas local transactions require only one. CPUs 2 and 3, on the other hand, undergo a slight decrease in throughput. This is because CPU 2 and 3 become more likely to collide with CPU 1 while trying to access the output port from switch 1 to switch 2 and memory section 3, respectively. In Fig. 6 b) the mean computational delay between consecutive memory accesses at each CPU was varied. As a performance measure, the ratio between actual measured throughput and the upper bound for the throughput (i. e. assuming instantaneous memory accesses) is depicted. CPU 1 and CPU 2 were assigned fixed remote memory access probabilities (P_{rma}) of 50% and 20%, respectively. The figure shows that longer delays between memory accesses lead to throughput values closer to the upper bound. This can be explained by the fact that collisions become less likely, and that the delay caused by a collision becomes smaller relative to the computational delay. It can also be seen that the increase in throughput occurs faster for CPUs that have a higher percentage of local memory accesses. As in the previous experiment, this is caused by the additional requests for switch output ports required for remote accesses. Modeling of this hierarchical network took about eight hours. Evaluation of the DSPNs took approximately half a second on the Linux PC mentioned in the previous section, whereas simulation required about 30 sec. on the same machine.

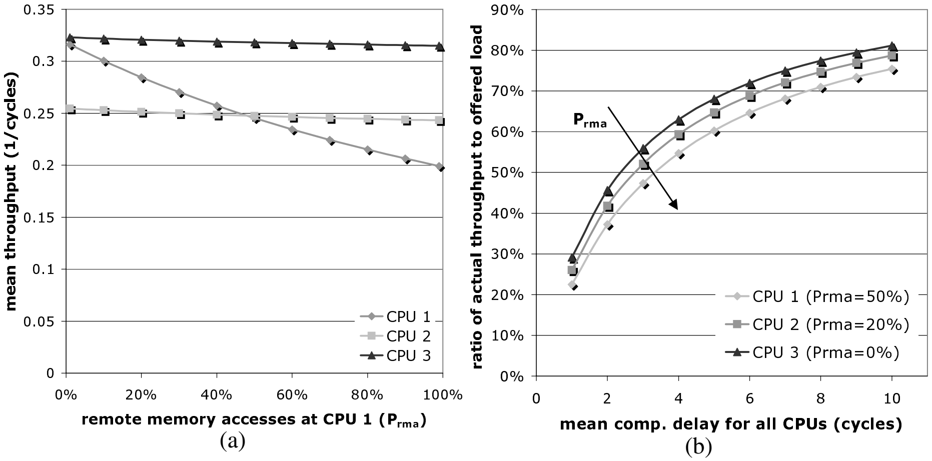


Fig. 6. Results for the modeling of the hierarchical NoC: a) mean throughput vs. percentage of remote memory accesses b) ratio of actual throughput to offered load vs. mean computational delay

This example shows how important design parameters of an NoC can be explored in an early stage of the NoC design flow by application of DSPN modeling techniques. For subsystems which require higher accuracy, successive refinement of the corresponding DSPN subnets can be applied to gain additional accuracy.

5 Conclusion

The DSPN modeling of basic NoC architectures has been presented in this paper. Applying Nios soft core processors which are connected via an Avalon communication structure it could be shown that the modeling results are very close to the values measured on an FPGA test bed. For this example it could also be shown that modeling effort can be efficiently traded for modeling accuracy. Quantitative results for modeling effort and computational complexity have been presented. Furthermore, a more complex hierarchical NoC has been modeled and the influence of parameters like the distribution of read and write accesses has been studied. This example shows that DSPNs can be leveraged for early stage modeling of communication processes.

References

1. Jantsch, A., Tenhunen, H.: Networks on Chip. Kluwer Academic Publishers (2003)
2. Nurmi, J., Tenhunen, H., Isoaho, J., Jantsch, A.: Interconnect Centric Design for Advanced SoC and NoC. Kluwer Academic Publishers (2004)
3. Kogel, T., Doerper, M., Wieferink, A., Leupers, R., Ascheid, G., Meyr, H., Goossens, S.: A modular simulation framework for architectural exploration of on-chip interconnection networks. In: CODES+ISSS. (2003) 7–12
4. Madsen, J., Mahadevan, S., Virk, K.: Network-centric system-level model for multiprocessor soc simulation. In Nurmi, J., ed.: Interconnect Centric Design for Advanced SoC and NoC. Kluwer Academic Publishers (2004)
5. Lahiri, K., Raghunathan, A., Dey, S.: System-level performance analysis for designing on-chip communication architectures. IEEE Trans. on CAD of Integrated Circuits and Systems **20** (2001) 768–783
6. Plosila, J., Seceleanu, T., Sere, K.: Network-centric system-level model for multiprocessor soc simulation. In Nurmi, J., ed.: Interconnect Centric Design for Advanced SoC and NoC. Kluwer Academic Publishers (2004)
7. Mickle, M.H.: Transient and steady-state performance modeling of parallel processors. Applied Mathematical Modelling **22** (1998) 533–543
8. Kleinrock, L.: Queueing Systems. Volume 1. John Wiley and Sons (1975)
9. Blume, H., von Sydow, T., Noll, T.G.: Performance analysis of soc communication by application of deterministic and stochastic petri nets. In: SAMOS. (2004) 484–493
10. Ciardo, G., Charkasova, L., Kotov, V., Rokicki, T.: Modeling a scalable high-speed interconnect with stochastic petri nets. In: Proc. of the Sixth International Workshop on Petri Nets and Performance Models PNPM'95. (1995) 83–94
11. Altera: Nios Embedded Processor Software Development Reference Manual. (2001)
12. Lindemann, C.: Performance Modeling with Deterministic and Stochastic Petri Nets. JOHN WILEY AND SONS (1998)
13. Petri net tools data base: http://www.daimi.au.dk/PetriNets. (2004)
14. DSPNexpress: http://www.dspnexpress.de. (2003)
15. Altera: Avalon: Bus specification manual, http://www.altera.com/literature/manual/mnl_avalon_bus.pdf. (2003)
16. Altera: SOPC Builder, http://www.altera.com/products/software/products/sopc/sop-index.html. (2004)

High Abstraction Level Design and Implementation Framework for Wireless Sensor Networks

Mauri Kuorilehto, Mikko Kohvakka, Marko Hännikäinen, and Timo D. Hämäläinen

Tampere University of Technology, Institute of Digital and Computer Systems,
P.O. Box 553, FIN-33101 Tampere, Finland
mauri.kuorilehto@tut.fi

Abstract. The diversity of applications, scarce resources, and large scale set demanding requirements for Wireless Sensor Networks (WSN). All requirements cannot be fulfilled by a general purpose WSN, but a development of application specific WSNs is needed. We present a novel WIreless SEnsor NEtwork Simulator (WISENES) framework for rapid design, simulation, evaluation, and implementation of both single nodes and large WSNs. New WSN design starts from high level Specification and Description Language (SDL) model, which is simulated and implemented on a prototype through code generation. One of the novel features is the back-annotation of measured values from physical prototypes to SDL model. The scalability and performance of WISENES have been evaluated with TUTWSN that is a very energy efficient new WSN. The results show only 6.7 percent difference between modeled and measured TUTWSN prototype energy consumption. Thus, WISENES hastens the development of WSN protocols and their evaluation in large networks.

1 Introduction

Wireless Sensor Networks (WSN) are the most demanding area of wireless communications. Challenges for WSNs are large scale, potentially constantly changing network topology, error prone environment, and demanding functional features. These features include multi-hopping, adhoc network creation, positioning, and autonomous operation with error recovery. Moreover, nodes should be small in size, which leads to limited processing and storage capacities. The energy should be scavenged from the environment or nodes should operate on batteries for a long time [1, 2].

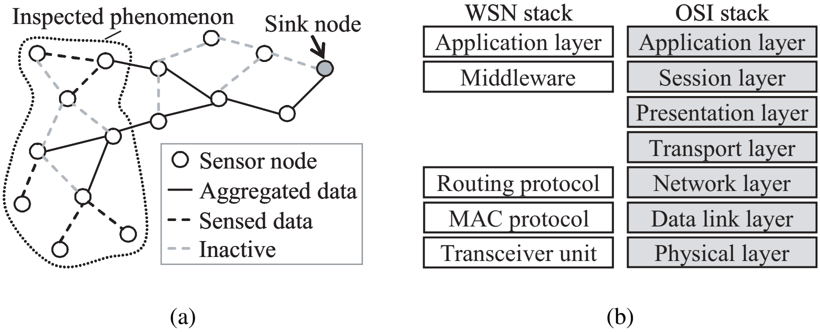
In a typical WSN scenario depicted in Fig. 1(a) nodes gather data around the inspected phenomenon, aggregate the data, and send them to a *sink node*. The communication is done through a layered protocol stack, an example of which is depicted in Fig. 1(b) in correspondence to the OSI reference model. A Medium Access Control (MAC) protocol on the data link layer manages the channel access and adhoc network topology. A routing protocol in the network layer creates multi-hop paths between nodes. Transport and presentation layers are typically omitted in order to reduce communication. A middleware abstracts underlying protocol and hardware inconsistencies from applications.

Due to the strict requirements a general purpose WSN is not feasible. This means that WSN protocols and node platforms must be configured to meet the requirements

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 384–393, 2005.

of the current application. The complexity of the design space [3] encourages the utilization of design automation tools and higher abstraction levels both for the design of a single node and the overall network. It is very important to consider all design issues at the same time, since a single unoptimized part can e.g. ruin the overall energy efficiency or lead to an excessive memory usage. This is often the case when protocol layers are developed independently.

The selection of an acceptable WSN design requires evaluation of different protocol and platform configurations. Hardware prototyping is suitable for a single node and small scale WSN testing (tens to hundreds of nodes). However, prototypes are not applicable for verifying a long term deployment of e.g. a 10,000-node network, but simulation tools are required. Pure functional network simulator, such as ns-2, is not feasible for WSNs because environmental constraints and physical phenomena must be considered.



(a) (b)

Fig. 1. A typical (a) WSN deployment and (b) WSN protocol stack in comparison to seven layer OSI-model

Current WSN design frameworks vary from low level emulators to high level modeling tools. Several low level node simulators [4, 5, 6] implement an environment for the simulation of applications above a widely used WSN Operating System (OS), TinyOS [7]. VisualSense [8] implements a framework for WSN application design and evaluation with Ptolemy II. It allows protocol and application design using different Models of Computation (MoC), but estimates the resource consumption in nodes only roughly. A comprehensive simulation framework for WSNs with accurate platform and resource modeling is implemented by SensorSim [9], but it does not support high level WSN design. The basic problem is that those based on low level tools are restricted in their scalability for large network configurations, whereas the high abstraction level frameworks do not give accurate information about the network and node performance.

Our WIreless SEnsor NEtwork Simulator (WISENES) is a novel, complete framework for the design, simulation, evaluation, and implementation of both nodes and WSNs. WISENES is based on our previous experience on automated HW/SW protocol implementations from high-level SDL (Specification and Description Language) and UML2.0 protocol models [10]. Unlike the other WSN design and simulation tools, WISENES uses high abstraction level for the protocol design but still outputs accurate results from the network and node performance. In addition, WISENES allows implementation of the final protocol stack through an automatic code generation.

WISENES has been used for the design and evaluation of our TUTWSN that currently has, to our best knowledge, the highest energy efficiency without sacrificing WSN functionality. A prototype network comprising tens of nodes exists. The key protocol is MAC that uses energy optimized Time Division Multiple Access (TDMA) for channel access. Compared to other TDMA or Carrier Sense Multiple Access (CSMA) WSN MAC protocols, e.g. [11, 12], TUTWSN achieves better energy efficiency due to the minimized idle listening times [13]. Results from WISENES prove the applicability of TUTWSN also for large scale networks.

This paper is organized as follows. Section 2 presents WISENES design and framework. In Section 3, TUTWSN design, implementation, and evaluation in WISENES are shown. Conclusions and future work are presented in Section 4.

2 WISENES Design Flow

Fig. 2 shows an overview of the WSN design with WISENES. The designer creates protocol layers and applications in SDL. A simulator is automatically generated for the evaluation of a single node and the network of nodes. In addition, code generation is used for the final executable for each node. WISENES does not support HW synthesis from the SDL, but creates interfaces for existing HW blocks on a node.

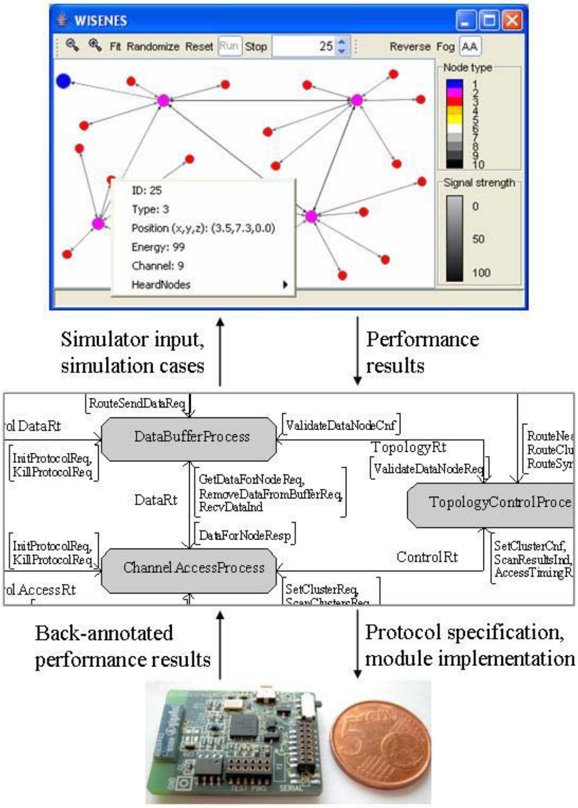
For simulations nodes are parameterized very accurately for WISENES in eXtensible Markup Language (XML). A unique feature in WISENES is the back-annotation of measured results from a physical prototype to the simulator. Measured values include e.g. the energy consumption per sequences of operations and even per an executed function. In this way WISENES combines the high abstraction level design and the accurate node performance evaluation into a single framework.

The basic steps in the flow are the creation of SDL model, functional simulations, the implementation of a limited scale prototype network, the back-annotation of the performance measurements for SDL model consistency checking and for improving simulator accuracy, and again simulations but now for a large scale network. Finally, the production ready WSN implementation is obtained. It is also possible to perform only simulations and based on that give constraints for the platform. This is useful when no physical platform is available or it is being designed.

2.1 WISENES Framework

A more detailed view to the WISENES framework is depicted in Fig. 3. The designer implements protocols and application tasks in SDL, produces XML configuration files, and integrates them to the WISENES framework. The framework consists of four main components, which are *central simulation control*, *transmission medium*, *phenomena channel*, and *sensor node*. WISENES outputs information both visually through a Graphical User Interface (GUI) and in detail to log files for postprocessing.

SDL in WISENES. The SDL hierarchy has multiple levels, of which *system level* consists of a number of *blocks* that clarify the representation. The behavior of a block is


WISENES
Fit Randomize Reset Run Stop
25
Reverse Fog AA
Node type
1
2
3
4
5
6
7
8
9
10
Signal strength
0
50
100
HeardNodes
Simulator input, simulation cases
Performance results
RouteSendDataReq
ol DataRt
DataBufferProcess
ValidateDataNodeCnf
TopologyRt
ValidateDataNodeReq
RouteNe
RouteClu
RouteSyn
InitProtocolReq, KillProtocolReq
GetDataForNodeReq, RemoveDataFromBufferReq, RecvDataInd
DataRt
TopologyControlProce
DataForNodeResp
InitProtocolReq, KillProtocolReq
ChannelAccessProcess
ControlRt
SetClusterCnf, ScanResultsInd, AccessTimingR
ol AccessRt
SetClusterReq, ScanClustersReq
Back-annotated performance results
Protocol specification, module implementation

ID: 25
Type: 3
Position (x,y,z): (3.5,7.3,0.0)
Energy: 99
Channel: 9

Fig. 2. WISENES design steps and their relations

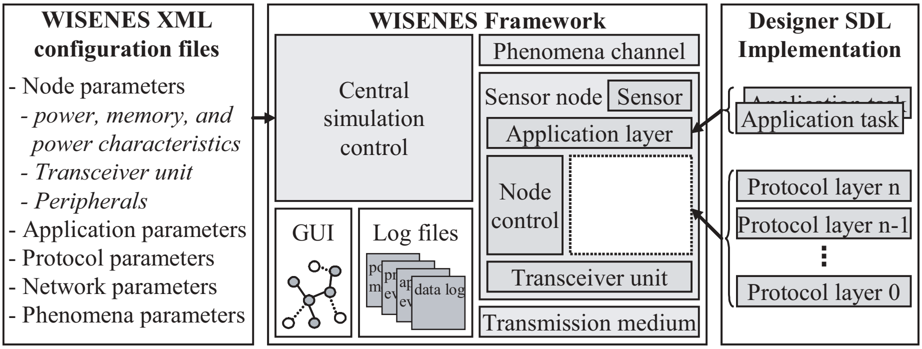


Fig. 3. User input and WISENES framework components

implemented in a number of *processes* described by Extended Finite State Machines (EFSM). A part of process functionality can be implemented as a *procedure* that is implemented by EFSM or in low level programming language. SDL processes communicate by asynchronous *signals* or synchronously by calling *remote procedures*.

We utilize Telelogic TAU SDL Suite 4.4 for the graphical SDL design and c-code generation. WISENES uses a discrete event simulation engine, in which events are processed in order of occurrence. The time concept is fully parallel allowing realistic modeling of multiple nodes.

Framework Components. WISENES framework components, except GUI, are implemented in SDL. The central simulation control parses input parameters, relays parameters to sensor nodes, maintains logs and GUI, and finishes simulations. The transmission medium models signal propagation in the wireless medium. A transmission success, failure, or attenuation is determined by the signal attenuation graph, which is specific to a transceiver unit. The phenomena channel models physical quantities that are measured by sensors. The communication between the framework components is implemented as SDL signals.

The sensor node is a dynamic block that has as many instances as there are nodes in a simulation. The sensor node has four framework components that implement interfaces for protocols and application tasks. The *transceiver unit* interfaces the transmission medium and models the power consumption of the transmitter and the receiver. The *sensor* models phenomena sensing and consumes energy while Analog-to-Digital Converter (ADC) and physical sensors are active. The *application layer* schedules and issues application tasks.

The *node control* is divided into two subcomponents. *NodeOSCtrl* implements OS routines that control execution scheduling, memory usage, and node sleep states and activation. *NodeSimulationCtrl* implements a per-node interface to the central simulation control for input parameterization and GUI updating. Further, NodeSimulationCtrl manages node power model in remote procedures that are called from other framework components during processing and peripheral or transceiver unit activation.

WSN Protocol Stack. The composition of a protocol stack implemented by the designer and the interfaces between the protocols are not restricted. Predefined interfaces must be met at the upper interface to the application layer, and at the lower interface to the transceiver unit. Further, a control interface for initiation, shutdown, and OS routines must be implemented for the node control.

Each protocol layer is implemented as an SDL block and may consist of any number of processes. The processes communicate with each others and with processes on other protocol layers. Application tasks are implemented as SDL procedures, which must implement a WISENES specific call interface.

WISENES Input and Output. XML configuration files specify node platforms and their processing, memory, and power consumption characteristics accurately. The platform parameters are divided into four separate files. Peripheral and transceiver unit parameters define available components and their energy and operational details. Node types are parameterized in a file referencing transceiver unit and peripheral parameters.

Individual nodes are specified in a file that defines only their node type and position coordinates, which makes automatic creation of large network configurations easy.

Application tasks are implemented in SDL but their resource requirements are defined in the input parameters. Protocol parameters consist of designer defined parameters for each layer. The signal propagation in the transmission medium is defined in network parameters and modeled physical quantities in phenomena parameters.

During a simulation run the progress of the simulation and the network topology are depicted in GUI, shown in Fig. 2. Accurate performance evaluation is done by post-processing extensive logs that are gathered during simulations and stored into files. The log information defines data processing at each layer, protocol and application events, network topology changes, transmission medium characteristics, and power, memory and processing capacity consumption for each node separately.

3 TUTWSN Implementation and Evaluation in WISENES

In this section we show an example how TUTWSN design and evaluation are carried out with WISENES. The target is a clustered network that consists of headnodes with full WSN functionality and subnodes that only transmit sensor data to a headnode. TUTWSN utilizes TDMA for intra-cluster communication and for data transfers between headnodes. Frequency Division Multiple Access (FDMA) is used to interleave nearby clusters to the available frequency band. This scheme and the very low activity times due to energy saving set strict timing requirements for the MAC protocol.

An environmental monitoring application is considered. It consists of two main tasks, temperature sensing and aggregation. All nodes acquire temperature in their surroundings and send the value to the cluster headnode. Headnodes aggregate data and send them to a sink node.

TUTWSN protocol stack in WISENES implements the MAC protocol and a routing protocol. The MAC protocol is divided into four parts. These are headnode functionality, subnode functionality, inter-cluster scanning, and data buffering. The routing protocol broadcasts active routing requests to its neighbors except the one it received the request from. The responses are sent selectively to those nodes pending for the route.

3.1 Prototype Platform

For the small scale network evaluation one of our existing TUTWSN prototypes depicted in Fig. 2 is chosen. The MicroController Unit (MCU) is a 2 MIPS Xemics XE88LC02 consisting of a CoolRisc 816 processor core, a 16-bit ADC, 22 KB program memory, 1 KB of data memory, and 8 KB EEPROM.

The transceiver unit is a 2.4 GHz NordicVLSI nRF2401 with 1 Mbps data rate. An integrated 16-bit CRC error detection is utilized by the TUTWSN MAC protocol. The energy for the prototype is supplied by a 0.22 F capacitor.

3.2 Code Generation from WISENES to Prototypes

A final executable is generated from a bounded SDL module that can vary from the complete protocol stack to a single SDL process implementing a part of a protocol layer. We use complete TUTWSN MAC protocol module as an example. The module is first detached from WISENES and then the C code is generated.

The generated code requires an SDL kernel for state, processing, and signaling control. A more lightweight kernel than the standard Telelogic TAU SDL Suite kernel, which is used in simulations, has been implemented for the prototypes. Our lightweight kernel implements the state control, processing scheduling, and signal exchange within the SDL module and a set of SDL library functions, e.g. for supporting standard SDL data types.

The module is connected to the environment with customizable input and output functions. In addition, the lightweight kernel includes the functions of NodeOSCtrl. These either interface OS routines, if available on the prototype, or offer restricted functionality. For the TUTWSN MAC protocol module, small scale controllers for memory and node state are implemented.

The lightweight kernel integrates the module to the rest of the system, which is then compiled and linked to the executable binary. The binary is then programmed to the nodes, one by one. Due to the required kernel functionality and the inefficiencies in the automatic code generation, the binary size for the TUTWSN MAC protocol is around 115K instructions.

The resources in our prototype are very scarce, which means that the final protocol and application implementations after simulations must be carried out manually. The optimized C and assembly implementation take 5.4K instructions. However, the optimized implementation is done only once per design and only for this kind of platforms with very limited resources.

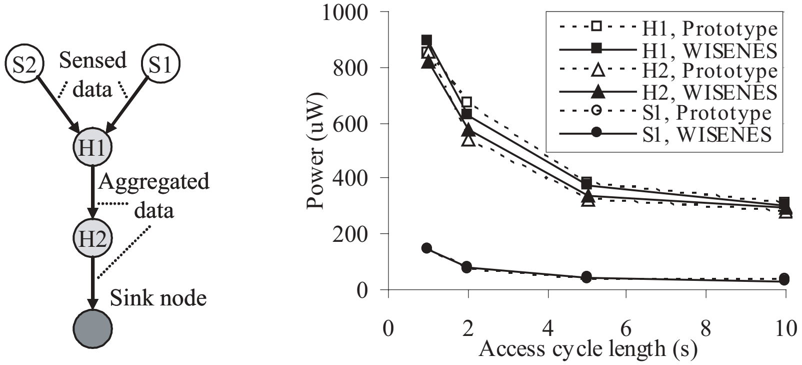
3.3 Prototype Modeling in WISENES

We refer the detailed modeling of platforms in the XML configuration parameters to as *prototype mapping*. The modeled aspects in the prototype mapping are timing and power, memory, and processing capacity consumption. The power consumption modeling concerns the activation times of peripherals, transceiver unit, and MCU. The memory consumption is handled by a memory controller implemented in NodeOSCtrl. For processing the protocols reserve time slots by issuing a remote procedure on NodeOSCtrl. The timing is modeled accurately by generating delays during the execution, the transceiver unit and peripheral access, and the signal propagation in the transmission medium.

For evaluating the accuracy of prototype mapping, similar test environments are constructed with prototypes and in WISENES. The measured nodes from the statically defined network topology are presented in Fig. 4(a). The subnodes (S1, S2) send their sensor reading once in an access cycle to the headnode H1, which aggregates and routes the data to the sink node through the headnode H2. We present results only for power consumption as the mapping of other aspects is straightforward.

Simulated and prototype power consumption results are depicted in Fig. 4(b) for S1, H1, and H2 with 1, 2, 5, and 10 second access cycles. The results are closely analogous, overall average difference being 6.73%. The averaged differences are 4.05% for H1, 4.80% for H2, and 11.34% for S1. The main reason for the less precise accuracy of

S1 is the slight timing inaccuracies in the WISENES modeling of node state changes. As the activity and the power consumption of subnodes are low, the state changes are crucial on overall power consumption.

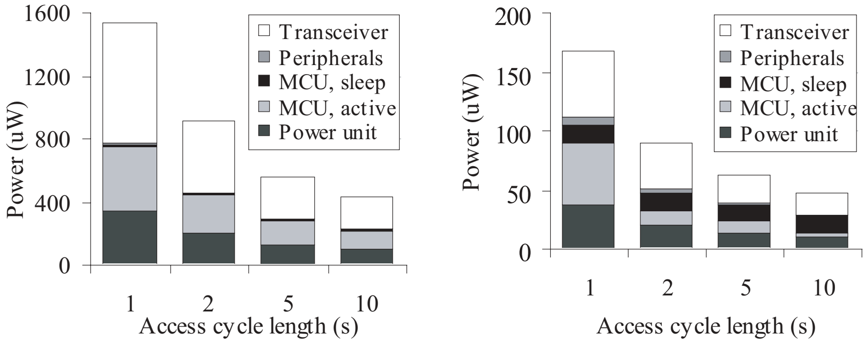


(a) (b)

Fig. 4. (a) Static topology and (b) power consumption results for prototype mapping accuracy evaluation

3.4 TUTWSN Performance in Large Scale Networks

A 10,000-node deployment on a 400m x 400m area is simulated to evaluate TUTWSN performance in large networks. Ten sink nodes are evenly scattered within the area. All nodes are able to act as both headnodes and subnodes. Their ratio at the beginning is 1:9. The most demanding simulations with 1 second access cycle lasted 21 days and produced 45 GB of log data. The average power consumption of different components calculated over ten arbitrarily selected headnodes and subnodes are depicted in Fig. 5 (a) and (b), respectively.



(a) (b)

Fig. 5. Average power consumption in 10000 node TUTWSN simulations for (a) headnode and (b) subnode

Fig. 5(a) clearly shows that the transceiver unit consumes most energy in the headnodes. As MCU must be active while the transceiver unit is powered up, its share is large. Due to the limited gain and leakage currents, also power unit is a major energy consumer. In the subnodes, the power consumption of transceiver unit is not as notable as in the headnodes, as shown in Fig. 5(b). All components are quite even due to the low activity of the subnodes. The scale of the vertical axis is eight times larger in Fig. 5(a).

The lifetimes of TUTWSN are 3.41, 5.07, 8.81, and 10.95 hours, when the access cycle is 1, 2, 5, and 10 seconds respectively. The network lifetime is considered as the time until half of the nodes have run out of energy. The reason for the short lifetimes is the small capacity of the capacitor. For comparison, with two serially connected AA batteries, the corresponding lifetimes are approximately 11, 16, 29, and 36 months.

4 Conclusions and Future Work

WISENES implements a complete flow for rapid WSN design and implementation. The high abstraction level modeling eases the design and evaluation of new WSNs, and final software executables for nodes are obtained through automatic code generation. Moreover, WISENES can be used to define constraints to node platform design for different application and protocol configurations.

WISENES has been implemented and it is fully functional. Results show that the modeling of node resource consumption is very accurate. Function-level energy consumption information is achieved from SDL designs.

Our future research on TUTWSN focuses on automated application distribution. The prototype mapping accuracy in WISENES will be further refined. The goal is to extend platform energy consumption modeling to instruction level. It would be a very challenging to optimize the use of instructions based on overall network requirements.

References

1. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: A survey on sensor networks. IEEE Communications Magazine **40** (2002) 102–114
2. Stankovic, J.A., Abdelzaher, T.F., Lu, C., Sha, L., Hou, J.C.: Real-time communication and coordination in embedded sensor networks. Proceedings of the IEEE **91** (2003) 1002–1022
3. Römer, K., Mattern, F.: The design space of wireless sensor networks. IEEE Wireless Communications **11** (2004) 54–61
4. Levis, P., Lee, N., Welsh, M., Culler, D.: Tossim: accurate and scalable simulation of entire tinyos applications. In: Proc. 1st ACM Conference on Embedded Networked Sensor Systems, Los Angeles, USA (2003) 126–137
5. Perrone, L.F., Nicol, D.M.: A scalable simulator for tinyos applications. In: Proc. Winter Simulation Conference 2002, San Diego, USA (2002) 679–687
6. Karir, M., Polley, J., Blazakis, D., McGee, J., Rusk, D., Baras, J.S.: Atemu: A fine-grained sensor network simulator. In: Proc. 1st IEEE International Conference on Sensor and Ad Hoc Communication Networks, Santa Clara, USA (2004) 145–152

7. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: Proc. 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, USA (2000) 94–103
8. Baldwin, P., Kohli, S., Lee, E.A., Liu, X., Zhao, Y.: Modeling of sensor nets in ptolemy II. In: Proc. 3rd International Symposium on Information Processing in Sensor Networks, Berkeley, USA (2004) 359–368
9. Park, S., Savvides, A., Srivastava, M.B.: Simulating networks of wireless sensors. In: Proc. Winter Simulation Conference 2001, Arlington, USA (2001) 1330–1338
10. Kukkala, P., Riihimäki, J., Hännikäinen, M., Hämäläinen, T.D., Kronlöf, K.: Uml 2.0 profile for embedded system design. In: Proc. 8th Design, Automation and Test in Europe Conference, Munich, Germany (2005) 710–715
11. IEEE standard 802.15.4: Wireless medium access control (mac) and physical layer (phy) specifications for low-rate wireless personal area networks (lr-wpans) (2003)
12. Ye, W., Heidemann, J., Estrin, D.: Medium access control with coordinated, adaptive sleeping for wireless sensor networks. IEEE/ACM Transactions on Networking **12** (2004) 493–506
13. Kohvakka, M., Hännikäinen, M., Hämäläinen, T.D.: Energy-efficient mac protocol for a wireless sensor network. Unpublished (2004)

The ODYSSEY Tool-Set for System-Level Synthesis of Object-Oriented Models

Maziar Goudarzi and Shaahin Hessabi

Department of Computer Engineering, Sharif University of Technology, Tehran, I.R.Iran
gudarzi@mehr.sharif.edu, hessabi@sharif.edu

Abstract. We describe implementation of design automation tools that we have developed to automate system-level design using our ODYSSEY methodology, which advocates *object-oriented (OO) modeling* of the embedded system and *ASIP-based implementation* of it. Two flows are automated: one synthesizes an ASIP from a given C++ class library, and the other one compiles a given C++ application to run on the ASIP that corresponds to the class library used in the application. This corresponds, respectively, to hardware- and software-generation for the embedded system while hardware-software interface is also automatically synthesized. This implementation also demonstrates three other advantages: firstly, the tool is capable of synthesizing polymorphism that, to the best of our knowledge, is unique among other C++ synthesizers; secondly, the tools generate an executable co-simulation model for the ASIP hardware and its software, and hence, enable early validation of the hardware-software system before full elaboration; and finally, error-prone language transformations are avoided by choosing C++ for application modeling and SystemC for ASIP implementation. [1]

1 Introduction

Embedded systems are all around us. They are getting more complex and we are getting used to using them in all aspects of our everyday life. More complex devices in lighter packages with longer battery life-time are demanded by consumers who would change their device for a new one in a few months and who expect to see prices falling everyday. Such complex embedded systems can no longer be designed in an ad hoc manner; *automated* design processes starting from *higher levels of abstraction* are essentially required. This is confirmed by emergence of Electronic System-Level (ESL) design as next level of abstraction in the move toward higher levels [1] and the studies that show current system-based design flows still incorporate several manual and time-consuming steps [2]. Addressing these requirements and shortcomings requires *system-level design methodologies*, to define the design flow, along with *tool chains*, to implement the methodology and facilitate manipulation and use of its artifacts.

In the ODYSSEY methodology [3], summarized in Section 2, we suggest to start design of embedded systems from an object-oriented (OO) model and to implement

[1] This work is supported by a research grant from the Department of High-Tech. Industries, Ministry of Industries and Mines of the Islamic Republic of Iran.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 394–403, 2005.

them as software running on Application-Specific Instruction-set Processors (ASIPs) customized to that embedded application. This provides embedded system designer with the reuse, flexibility, and complexity management advantages of OO modeling along with the reusability and extendability of ASIP (as opposed to ASIC) for system implementation.

The methodology [4, 5] treats OO designs in general, with no specific OO language or tool in mind. To implement it, however, specific languages and tools need to be chosen. We decided to use a single language for hardware and software parts of the system so as to avoid error-prone language transformations. Such transformations comprise a well-known source of many difficulties and errors, in developing system-level design-automation tools, that arise due to semantic difference between languages as well as complicated tool operations. Further to the above single-language requirement, the chosen HDL must be synthesizable by available tools to enable us report experimental results of complete implementation in addition to simulation results. Three major alternatives were OO variants of VHDL, Java, and C++. Several extensions to VHDL exist [6, 7, 8, 9] that include OO constructs, but they differ in their interpretation of OO concepts and only a few of them provide a path to synthesis. Hardware synthesis from Java [10, 11, 12] is also reported in the literature. However, its reported synthesis tools are academic and their level of support for various language constructs were unknown to us. We finally chose C++ since it offers necessary modeling features, it is widely used by system designers for early system modeling and validation, and its SystemC class library is a synthesizable HDL that is executable and provides high simulation efficiency.

In the rest of this paper, the big picture of our system-level design methodology is presented in next section. Section 3 provides our proposed synthesis flow from C++ to final implementation. Section 4 presents the details of our system-level synthesizer tools. Testing procedure of the tool and system-synthesis experiments are given in Section 5 and finally Section 6 concludes the paper.

2 The ODYSSEY System-Level Design Methodology

Our system-level design methodology, named ODYSSEY (Object-oriented Design and sYntheSiS of Embedded sYstems) [3], consists of two parts: a *modeling methodology* and an *implementation methodology*. In modeling the system-under-design, we advocate the OO methodology, while for system implementation it follows the idea of *programmable* platforms. The rationale behind our modeling methodology is the dominance of software over hardware in embedded systems [13] and the high reputation of OO in the software design community due to its good support for design abstraction and reuse. As the motive for our implementation method, the high cost of design along with the high cost and risk of manufacturing application-specific integrated circuits (ASICs) in today very deep submicron technologies [14] justifies our decision.

ODYSSEY wishes to view the system model as a collection of concurrently communicating objects (application-level objects, not implementation-level ones). The model of computation is objects communicating through structured messages. Multiple threads of control can co-exist in the system. Guarded operations are considered for synchronization between concurrent threads.

To implement such a system model, a two-level mapping is performed. First, the concurrency in the system model is mapped to a network of interconnected processors. Then, each processor is specialized to the set of objects that it is to serve by moving some functionality to hardware; we call such a specialized processor an object-oriented application-specific instruction processor, or an OO-ASIP[2]. The quantum of this migration to hardware, and hence the hardware/software partitioning quantum in ODYSSEY, is a *method* of a class; methods assigned to the hardware partition are called *hardware methods* and the others are *software methods*. We have previously presented two architectures for an OO-ASIP [4, 15]. The latter gives a scalable efficient implementation for virtual method dispatch to hardware- as well as to software-methods and is implemented by our tool; however, it necessitates additional care during synthesis that is discussed in Section 4.2.

As mentioned above, we have broken implementation of the ODYSSEY design flow into two parts: mapping the objects to the processors, and implementing the thread(s) assigned to each processor. In this paper, we present the latter where a single processor is to be synthesized and a single thread of execution is considered.

3 Single Processor Synthesis Flow

Fig. 1 shows the synthesis flow for a single-processor target. The input program is given as a set of header and program files that together define the class library as well as the `main()` function where the objects are instantiated and the sequence of method calls among them is specified. The entire process is divided into two layers: we consider the upper layer as *system-level synthesis*; this layer takes the system model and produces the software- along with the hardware-architecture in a mix of structural and behavioral modeling styles. The lower layer is considered *downstream synthesis*; it takes the above-mentioned hardware and software partitions and produces gate-level hardware and object-code software. We focus on the system-level synthesizer in this paper. Downstream synthesis uses traditional tools and is not of particular interest here. It is noteworthy, however, that since we pass the hardware partition directly to downstream synthesis (so as to take advantage of the available SystemC-synthesis technology) the subset of C++ constructs that such tools accept defines the synthesizable subset that our tool allows in method code.

In system-level synthesis (the part above dashed line in Fig. 1), the input program and class library are parsed and analyzed to extract class-inheritance hierarchy, symbol table, and definitions of methods. Then the methods are, currently manually, assigned to either hardware or software partitions (the "Partitioning" box in Fig. 1) and then the method definitions are transformed to suit their assigned partition (boxes labeled "Transformations"). Moreover, the partitioning box generates some macros (shown as "Instr-set extensions" box) that are required for proper compilation of the software

[2] The OO-ASIP does have other features than simple hardware acceleration. Interested reader is referred to [4, 15] for more details.

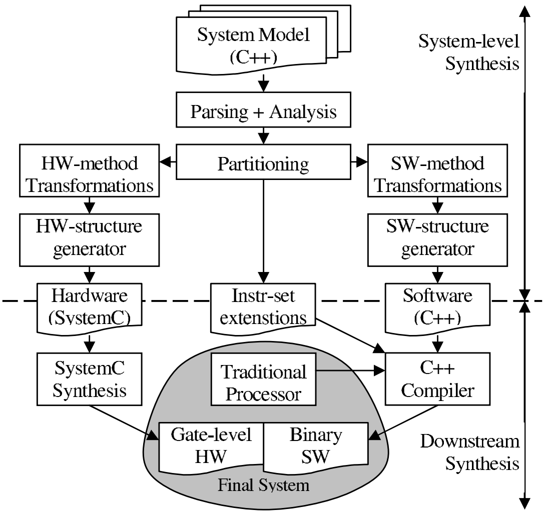


Fig. 1. The ODYSSEY Single-processor synthesis flow

partition; these macros represent hardware methods wherever they are called in the software partition. Transformed methods in each partition are then passed to their corresponding *structure generator* box that appropriately assembles them together. The resulting hardware (in SystemC) and software (in C++) can be simulated together, as the co-simulation of the post-synthesis system, in any environment that supports the SystemC class library.

4 System-Level Synthesis

The big picture of system-level synthesis and the correspondence between model and implementation elements are depicted in Fig. 2. The input consists of a set of classes along with a `main()` function (the left hand side of Fig. 2). The output is a synthesizable processor architecture comprised of some hardware units and a traditional processor core, along with a software architecture containing a set of software routines and a single `thread_main()` function (the right hand side of Fig. 2). The implementations of hardware methods are put in the hardware modules in the middle of the OO-ASIP box in Fig. 2, whereas the implementations of software methods are put in the “traditional processor” module. The input `main()` function is converted to the `thread_main()` routine in the processor software. The objects’ data are put in an object-identifier-addressable memory unit, referred to as *object management unit* or *OMU*.

4.1 Structure of the Tool

The synthesis tool is object-oriented itself. It consists of three major types of objects: an input parser object, an output writer object, and per partition synthesizer objects re-

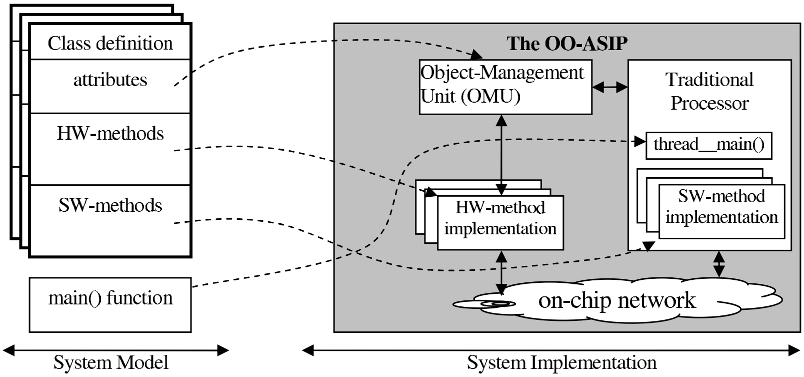


Fig. 2. The big picture of transformations in the system-synthesis process

siding between the previous two. The big picture of these objects and their functionality are briefly presented below. For ease of referencing, the following definitions are made:

Definition 1: The input OO model consists of a class library that declares and defines all available classes. We refer to this *class-library-under-synthesis* as *CLUS* hereafter.

Definition 2: The declaration of objects and the sequence of method calls among them is defined in a `main()` function in the input OO model. This *application-under-synthesis* is referred to as *AUS* hereafter.

Input Parser: This object reads in the input file and produces a parse tree along with other necessary data structures for further processing. If the input model contains several files, they are all concatenated into a single one for easier parsing. The input file contains both the CLUS and the AUS. At the construction time of this object, the input file is parsed, the hierarchy of classes is built, the symbol table is generated, and the member functions and their definitions are found. Then, the system is partitioned by assigning each method to the hardware or the software partition.

Synthesizers: These are the main objects responsible for synthesizing hardware and software methods. They operate on individual method definitions and transform them to suit hardware or software implementation according to their assigned partition. More details of these transformations are presented in Section 4.2.

Output Writer: This object assembles all hardware and software methods synthesized by above Synthesizer objects and writes out the complete co-simulation model of the OO-ASIP (see Section 4.3).

4.2 Model Transformations

Since SystemC is actually a C++ class library itself, support for several C++ constructs is provided effortlessly. However, some constructs need special handling and transformations, explained below, due to the special architecture of the OO-ASIP. This speciality arises from the partitioning quantum of our methodology; we assign each *class method* to either partition, and consequently, dispatching a call to method implementations, passing parameters to them, and returning values from them needs to be carefully worked out so that all four cases of hardware or software caller to hardware or software callee are appropriately handled. Furthermore, the methodology allows redefinitions of the same virtual method to reside in different partitions. This is a significant feature of the methodology that enables systematic application of software patches to extend, upgrade, or correct missing or faulty hardware units. To provide efficient dispatching of a virtual method call to hardware- as well as software-method implementations, we have devised a network-based dispatching mechanism presented in [15]; we view each method call as a packet to be sent over a network from the caller to the callee, carrying the call parameters as its data payload. This approach necessitates a special object numbering scheme, instead of the traditional address-in-memory scheme, that in turn necessitates special treatment of object instantiation and pointer-to-object declaration and use. The following paragraphs discuss each of the above special handlings and presents the corresponding transformations applied to the source C++ routines.

Object Instantiation. In a traditional processor, each object is assigned a memory portion whose starting address identifies the object. In the same way, we also allocate memory portions for object data; however, our special object-numbering scheme requires each object to be identified by a new object-identifier comprised of the identifier of the class of the object (i.e. *cid*) and the unique-in-this-class number assigned to this object (i.e. *objn*). We refer to this *(cid, objn)* pair as *oid* hereafter. This change in the object identifier introduces two issues: allocating a unique *oid* per object, and mapping it to the memory address when the object data is to be accessed. The first issue is handled by keeping track of per-class *objn* numbers already assigned to objects. The second issue is handled by a mapping hardware (the OMU) that translates the *oid* to its physical address. In other words, the OMU is an *oid*-addressable memory.

Object-Pointer Declaration and Use. The change we have proposed in the object identifier results in a change in the values that the pointers hold; a pointer holds *oid* values now. In transforming a class method (regardless of its being hardware- or software-method), wherever the address of an object is assigned to a pointer, the *oid* of that object is replaced instead.

Access to Object Attributes. To access the attributes of an object, the physical address of the object data in memory must be known. This was readily available in traditional processors as the object identifier, but in our methodology this needs to be extracted from the *oid*. To do this, our synthesis tool identifies all object access statements in the input program and replaces them with an `OBJ_ATTR( oid, attr_index )` macro in the output. The first operand of the macro is the *oid* of the desired object, which is translated by the macro to the starting address of the object, and the second operand is the index of the desired attribute in the object data storage.

Virtual Method Calls. To invoke the packet-based method-dispatching mechanism in hardware- and software-methods, the virtual method calls in the input program are replaced by special macros of VMC_BY_HW and VMC_BY_SW respectively. Both macros take two parameters; the first parameter is the *oid* of the called object and the second one is the identifier of the called method.

Passing Parameters to Virtual Methods. The same packet that dispatches a method call can indeed carry the parameters of the call. Simple data types, e.g. int, char, float, are trivial in this regard; however, more interesting cases arise when considering complex data types such as objects, arrays, pointers, and references. Objects and arrays can be passed by value, but this incurs an overhead due to transfer of possibly large amounts of data between the caller and the callee. Call by reference, and also pointer parameters, are only supported for objects since only objects are stored in the global data memory that is equally accessible to all hardware and software method implementations; if pointer parameters and/or call-by-reference are desired for other data types, the actual parameters can be declared as objects instead.

Returning Values from Virtual Methods. As in C++ programs, method calls are blocking in the system implementation; i.e. the caller waits for the callee to finish its operation. This end-of-operation is announced by a `METHOD_DONE` packet from the callee to the caller. This same packet can also carry the return value. All `return` and `return(val)` statements in method definitions are respectively replaced by `RETURN` and `RETURN_VAL(val)` macros that accomplish the necessary packet assembly and posting.

4.3 The Co-simulation Model

Our tool-set not only generates the hardware and software partitions, but also produces a simulatable post-synthesis model to allow integration-test before completion of downstream synthesis steps. This co-simulation model is in SystemC and contains hardware- as well as software-methods in their transformed C++ form; consequently, it not only provides integration-test earlier in the synthesis process, but also runs much faster than detailed post-downstream-synthesis co-simulation. This enables the designer to verify the system-synthesis process in isolation from the downstream process.

The architecture of the co-simulation model is shown in Fig. 3. The OO-ASIP module has two network lines, `net_in` and `net_out`, to connect to other OO-ASIPs on a network. The `reset` input is used to restart the OO-ASIP. The `clk` input is required by the SystemC synthesis tool (it only supports SC_CTHREAD, i.e. clocked thread, processes for synthesis) and could be omitted if only co-simulation were desired. Assertion of the `reset` signal starts the co-simulation. This triggers `restarter()` process which invokes the `thread_main()` routine. This routine first calls constructors of the objects, and then continues as in the original `main()`. To make a method call, the `thread_main()` routine assembles a packet, sends it to the `net_out` output of the `cpu` module, and waits for the method completion. In our current implementation, `net_in` and `net_out` ports of the OO-ASIP module are externally connected together while all hardware modules and the `cpu` itself are listening to the network and get invoked by any new packet sent there. Everybody checks the packet destination against its own address and takes it if they match. Each hardware module implements only one hardware method, whereas the `cpu` can contain several software methods. Thus, if the recipient is a hardware module, its single transformed-by-the-tool method code is run; but if the

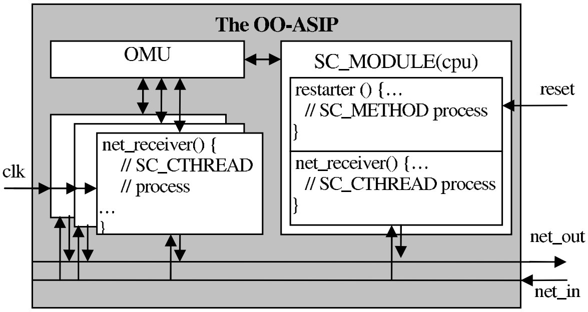


Fig. 3. The architecture of the post-synthesis co-simulation model

cpu receives the packet, it is internally dispatched to the appropriate software routine implementing the called method. Every hardware or software method may also call a virtual method; the procedure of method dispatching is the same as non-virtual calls. All software and hardware methods may also access attributes of an object; this is redirected to the OMU which implements an object-identifier-addressable memory.

5 Experiments

We implemented our system-synthesizer tools in C++; the implementation, in its current state, consists of about 3000 lines of code. It has taken about two man-months to devise the co-simulation model and manually develop and test it and an additional 5 man-months to develop and test the tools.

Table 1. Summary of the test cases devised for the tools

Test	Description
test1	Object instantiation + constructor invocation.
test2	Pointer declaration, assignment, and use.
test3	Virtual method call (VMC) by the `main()` function. The call is dispatched to a hardware-method, it finishes, and then control is passed back to `main()`.
test4	The same as above, but to a software-method callee.
test5	VMC by the `main()` to a hardware method. This time both the hardware method and `main()` access and manipulate the object attributes.
test6	The same as above, but to a software-method callee.
test7	VMC by the `main()` while passing two parameters: a by-value parameter, and a by-reference object.
test8	The same as above, but to a software-method callee.
test9	VMC by the `main()` to a hardware-method that returns a value
test10	The same as above, but to a software-method callee.
tests 11-18	The same as *test3* to *test10* above, but this time a hardware method starts the transactions.

To thoroughly verify proper operation of the tool, we devised a comprehensive set of test cases that incrementally covered all aspects that the tool should handle (Section 4.2). Table 1 gives a summary of these test cases and the points that each adds to previous ones. The first two test cases deal with object instantiation and object-pointer declaration and use. Tests number 3 to 10 verify proper synthesis when a software caller makes a virtual method call (tests 3 and 4), when attributes of the object are manipulated by the caller and the callee (tests 5 and 6), when by-value and by-reference parameters are passed to the callee (tests 7 and 8), and finally when a value is returned by the callee (tests 9 and 10). This same set of tests are performed for a hardware caller by tests number 11 to 18.

The co-simulation capability of our generated hardware-software system enabled us to make sure that the system-synthesis is correctly performed on each test case by running the pre- and post-synthesis programs and comparing their outputs against each other. This same capability can benefit system-designers in verifying the system-synthesis process and validating the partitioned system before proceeding to downstream synthesis steps.

6 Summary and Conclusion

The main thrust of this paper is to describe and discuss implementation details and advantages of an EDA tool-set that automates design of embedded systems in the ODYSSEY methodology. These tools automatically generate hardware, software, and their interface from a given class library and application. Although the methodology is language-neutral, its implementation (i.e., the EDA tool-set) cannot stay neutral on this. We chose C++ as the input language and SystemC-C++ as the output for hardware and software components respectively; these choices eliminated error-prone language transformations. Furthermore, this paper introduced a hardware-software co-simulation model that our EDA tool-set generates as the result of system-level synthesis. This co-simulation model is in SystemC, and hence, is an *executable model* of the hardware-software partitioned system generated by the tools. This property was used in this paper to efficiently verify correct operation of our design automation tools by comparing the results of executing pre- and post-synthesis models. This same approach can be used to validate consistency of generated hardware and software partitions and their interface. This efficient validation, due to executable high-level co-simulation model, allows early detection of probable mistakes or inconsistencies and serves as a design checkpoint before diving into time-consuming and intricate process of elaborating hardware and software partitions.

We are currently working on downstream synthesis steps (see Fig. 1) to complete the automated tool-chain from concept to working implementation on an FPGA board so as to make it practically possible to realize embedded systems in ODYSSEY style. Real-life case studies will be conducted afterwards to evaluate, tune, and/or amend the tool chain and/or the methodology.

References

1. Flaherty, N.: On a higher level. The IEE Review (2004) 22–24 Report from the 41st Design Automation Conference.
2. Schubert, T., Hanisch, J., Gerlach, J., Appell, J., Nebel, W.: Evaluating a system-based design flow. IEE Electronics Systems and Software (2004) 29–33
3. ODYSSEY Project: Online Homepage. (2005) http://ce.sharif.ir/˜ odyssey.
4. Goudarzi, M., Hessabi, S., Mycroft, A.: Object-oriented ASIP design and synthesis. Proc. of Forum on specification & Design Languages (FDL) (2003) Frankfurt.
5. Goudarzi, M., Hessabi, S., Mycroft, A.: Object-oriented embedded system development based on synthesis and reuse of OO-ASIPs. Journal of Universal Computer Science (J.UCS) (2004) In press.
6. Ashenden, P., Wilsey, P., Martin, D.: SUAVE: Painless extension for an object-oriented VHDL. Proc. of VHDL Int'l Users' Forum (VIUF) (1997)
7. Radetzki, M.: Synthesis of Digital Circuits from Object-Oriented Specifications. PhD thesis, University of Oldenburg, Germany (2000)
8. Schumacher, G., Nebel, W.: Inheritance concept for signals in object-oriented extensions to VHDL. Proc. of EURO-DAC with EURO-VHDL (1995)
9. Radetzki, M., Putzke-Roming, W., Nebel, W., Maginot, S., Berge, J., Tagant, A.: VHDL language extensions to support abstraction and re-use. Proc. of Workshop on Libraries, Component Modelling, and Quality Assurance (1997)
10. OASE Project: Objektorientierter hArdware/Software Entwurf: Online Home Page. (2004) http://www-ti.informatik.uni-tuebingen.de/˜ oase/.
11. Kuhn, T., Oppold, T., Winterholer, M., Rosenstiel, W., Edwards, M., Kashai, Y.: A framework for object oriented hardware specification, verification, and synthesis. Proc. of Design Automation Conference (DAC) (2001) Las Vegas, Nevada.
12. Young, J., MacDonald, J., Shilman, M., Tabbara, A., Hilfinger, P., Newton, A.: Design and specification of embedded systems in Java using successive, formal refinement. Proc. of Design Automation Conference (DAC) (1998)
13. International Semiconductor Roadmap Committee: International Technology Roadmap for Semiconductors (ITRS)-Design. (2003) http://public.itrs.net.
14. Keutzer, K., Malik, S., Newton, A.: From ASIC to ASIP: the next design discontinuity. Proc. of Int'l Conference on Computer Design (ICCD) (2002)
15. Goudarzi, M., Hessabi, S., Mycroft, A.: Overhead-free polymorphism in network-on-chip implementation of object-oriented models. Proc. of Design Automation and Test in Europe (DATE) (2004) Paris.

Design and Implementation of a WLAN Terminal Using UML 2.0 Based Design Flow

P. Kukkala, M. Hännikäinen and T.D. Hämäläinen

Tampere University of Technology, Institute of Digital and Computer Systems,
P.O. Box 553, FIN-33101 Tampere, Finland
petri.kukkala@tut.fi

Abstract. This paper presents a UML 2.0 based design flow for real-time embedded systems. The flow starts with UML 2.0 application, architecture and mapping models for our TUTWLAN terminal with its medium access control protocol. As a result, the hardware/software implementation on Altera Excalibur FPGA is achieved. Implementation utilizes eCos real-time operating system, and hardware accelerators for time-critical protocol functions. The design flow is prototyped in practice showing rapid UML 2.0 application model modification, real-time protocol processing in an image transfer application, and execution monitoring.

1 Introduction

Several design frameworks have been proposed for high-level systems design to meet the challenges of ever-increasing system complexity. However, most of the approaches concentrate on a specific flow aspect, but rarely cover the whole flow from an abstract model to physical implementation.

UML 2.0 is converging on a general design language that can be understood by system designers as well as software and hardware engineers [8]. Many UML profiles have been proposed to adapt UML to this purpose. For instance, the UML Platform profile [2] introduces a way to model architecture resources and services, and the UML Profile for Schedulability, Performance, and Time [9] defines notations for building models of real-time systems with Quality of Service (QoS) parameters.

This paper presents a novel UML 2.0 based design flow that uses a custom UML profile, called TUT-Profile [7]. The flow comprises the UML application, architecture and mapping models, and includes automatic code generation, real-time execution monitoring, model profiling and performance back-annotation to the UML models. The main contribution of the approach are the methods to govern the whole flow using UML 2.0.

This paper shows the use of UML 2.0 in the implementation of a time-critical embedded system including both hardware and software. The target is the implementation of a proprietary TUTWLAN terminal on Altera Excalibur FPGA with an embedded processor core and eCos Real-time Operating System (RTOS). Rapid model modification is performed by changing the functionality of the terminal to illustrate the use of the design flow in practice.

The paper is organized as follows. First, TUTWLAN is introduced in Section 2. The design flow to implement the TUTWLAN terminal is presented in Section 3. Thereafter,

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 404–413, 2005.

the implemented terminal, real-time execution monitoring, and rapid model modification are presented in Section 4. Finally, Section 5 concludes the paper.

2 TUTWLAN and the TUTMAC Protocol

TUTWLAN [6] is a proprietary WLAN developed at Tampere University of Technology (TUT). TUTWLAN solved the problems of scalability, QoS and security present in standard WLANs. The wireless network has a centrally controlled topology, where one base station controls and manages multiple portable terminals. Several configurations have been developed for different purposes and platforms. In this we present one configuration of the TUTMAC protocol.

TUTMAC is a dynamic reservation Time Division Multiple Access (TDMA) based Medium Access Control (MAC) protocol for TUTWLAN. The protocol supports negotiated QoS parameters for the data transfer, and is capable for managing the dynamically changing network topology.

The protocol contains functions for Cyclic Redundancy Check (CRC) and encryption. CRC is performed for headers with CRC-8 algorithm, and for payload data with CRC-32 algorithm. Encryption is performed for payload data using an Improved Wired Equivalent Privacy (IWEP) algorithm [5]. The algorithm encrypts payload data in 64-bit blocks, and uses an encryption key of same size.

The functions are performed for every packet sent and received by a terminal. Thus, their performance become significant, especially, when the data throughput increases and several packets are simultaneously processed by the protocol. Depending on the implementation, the algorithms may need hardware acceleration to achieve adequate delays for data. Further, the radio channel access has to maintain accurate frame synchronization in the TDMA scheduling, which sets tight real-time constraints and need for prioritizing the protocol processing.

3 UML 2.0 Design Flow for the TUTWLAN Terminal

The flow to design and implement the TUTWLAN terminal is presented in Fig. 1. The flow starts with three UML models for an application, architecture and mapping, which describe the TUTWLAN terminal as a whole. At least an application must be provided by a designer, but the architecture and mapping can also be produced using an architecture exploration tool, such as Koski [10].

The *application model* implements the functionality of the terminal. Software for TUTMAC is automatically generated based on the application model. External C functions can be included in the model if needed. Software is executed using an RTOS, and the thread configuration is retrieved from the *mapping model*. In this case, the thread configuration is fixed and manually retrieved, but the task can also be automated. The *architecture model* is composed of existing hardware components available in a library. In this case, the implementation is fixed, but the component selection can also be dynamic and automated.

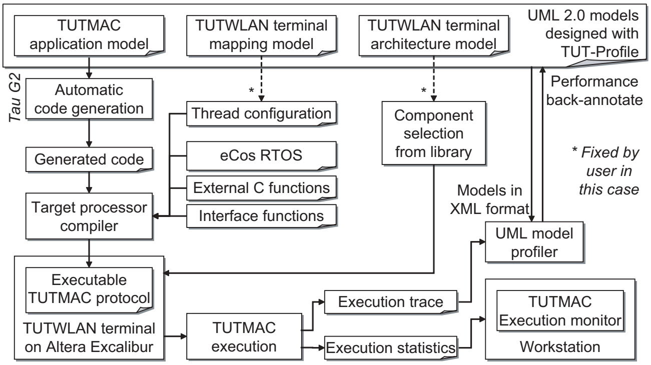

TUTMAC application model
TUTWLAN terminal mapping model
TUTWLAN terminal architecture model
UML 2.0 models designed with TUT-Profile
Tau G2
Automatic code generation
Generated code
Target processor compiler
Thread configuration
eCos RTOS
External C functions
Interface functions
Component selection from library
Performance back-annotate
* Fixed by user in this case
Models in XML format
UML model profiler
Executable TUTMAC protocol
TUTWLAN terminal on Altera Excalibur
TUTMAC execution
Execution trace
Execution statistics
TUTMAC Execution monitor
Workstation

Fig. 1. UML 2.0 based design flow for the implementation of the TUTWLAN terminal

During execution, a trace and statistics are collected using custom functions. The model profiling uses the trace to back-annotate performance information to the UML models. Statistics are used for the real-time execution monitoring.

The terminal is implemented on Altera Excalibur FPGA on EPXA1 development board [1]. The FPGA contains an ARM9 processor and a Programmable Logic Device (PLD) connected by AMBA Bus (AHB) and dual-port memory. A 2.4 GHz MACless Intersil HW1151-EVAL radio transceiver is connected to the development board with an expansion header. On PLD, custom hardware accelerators and interfaces to access external devices are implemented using VHDL.

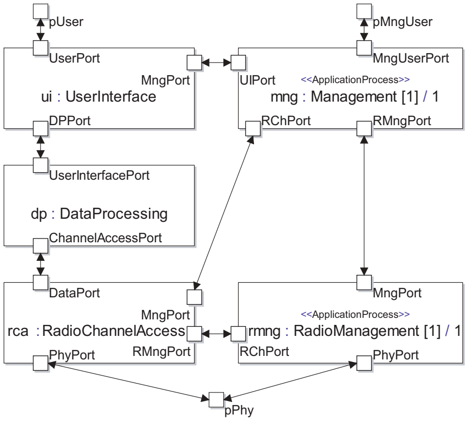
Telelogic Tau G2 [11] is used as an UML design tool. The tool also enables automatic C code generation for an application. The generated code supports POSIX standard for threaded execution of software. eCos RTOS [3] is utilized to the software execution. eCos provides a POSIX compatibility layer supporting the standard Application Programming Interface (API) to interface the kernel.

3.1 UML 2.0 Models

The UML models are constrained by our custom TUT-Profile [7], which is targeted to the design of real-time embedded systems using a model based approach. The profile defines a set of stereotypes for extending the standard UML metaclasses, and design practises to describe an application and architecture as well as their mapping. The stereotypes have tagged values for the real-time constraints.

TUTMAC Application Model. The class hierarchy of TUTMAC is designed using *class diagrams*. Thereafter, *composite structure diagrams* are used to describe the structure of the protocol in a more detailed way. The *parts* (class instances) communicate sending *signals* via *ports* and *connectors*.

The composite structure diagram of the top-level class `Tutmac_Protocol` is presented in Fig. 2(a). The `mng` and `rmng` parts are instances of the functional components, and they represent the processes of the application model. The `ui`, `rca`, and `dp` parts are instances of the structural components, which are further hierarchically modeled using class diagrams and composite structure diagrams.



(a) Top-level composite structure diagram

Diagram type	*Amount*
Class diagrams	18
Composite structure diagrams	5
Statechart diagrams	41

Property	*Amount*
State machines (processes)	20
Ports	52
Signal types	40

(b) Amount of diagrams and properties

Fig. 2. TUTMAC UML 2.0 application model with TUT-Profile

In TUTMAC, the behavior of functional components is expressed using statechart diagrams combined with the UML 2.0 textual notation. Statecharts are asynchronous communicating Extended Finite State Machines (EFSM) [4].

Figure 2(b) presents the amount of UML diagrams and main properties of the TUTMAC UML model to illustrate the size and complexity of the model.

TUTWLAN Terminal Architecture Model In this case, the architecture of the TUTWLAN terminal is fixed. An existing library contains UML models for the fixed components of Excalibur FPGA, including the processor, AHB and dual-port memory, as well as custom hardware accelerators and interfaces. The components are stereotyped using TUT-Profile to parameterize each component. The architecture model for the TUTWLAN terminal is presented in Fig. 3.

TUTWLAN Terminal Mapping Model The mapping model connects an application with an architecture. Mapping is performed in two stages. First, application processes are grouped, and thereafter, the groups are mapped to an architecture. Grouping can be performed according to different criteria, such as workload distribution, communication between groups, and the size of groups.

A process group maps a number of application processes to a platform component instance. In software, a process group is implemented as a single thread, and in hardware, as a single hardware accelerator, depending on the case.

Figure 4 shows the process grouping, which is based on the features of the processes. Most of the processes are divided into `LowPrio` and `HighPrio` groups that correspond to low and high priority threads in the software implementation. In addition, the processes related to the CRC-32 calculation and encryption are grouped into separate `CRC32` and

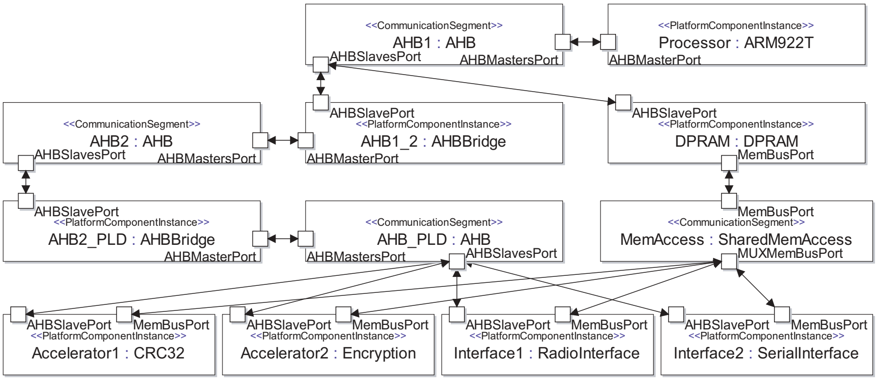


Fig. 3. TUTWLAN terminal architecture

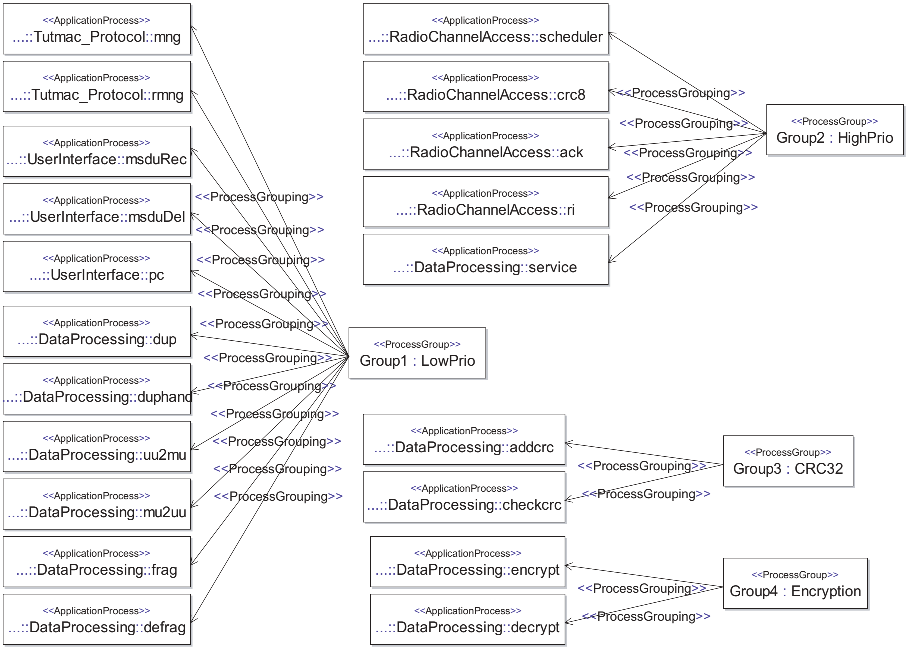


Fig. 4. TUTMAC process grouping

`Encryption` groups. It should be noted that the flow does not restrict the number of groups.

Figure 5 presents the platform mapping of TUTMAC. The process groups defined above are mapped on the architecture model. In this case, the mapping model is created manually, but the task can be automated using an architecture exploration tool, such as Koski [10]. If the designer has created a preliminary mapping when using architecture exploration, the tool optimizes the mapping, but retains fixed mappings.

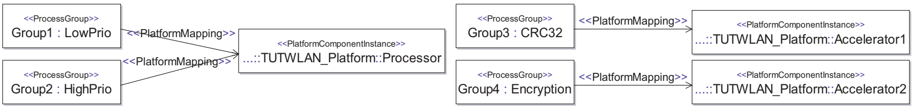


Fig. 5. TUTMAC platform mapping

3.2 Implementation Phase

C code for TUTMAC is automatically generated based on the application model. The generated code implements the functionality of state machines, but not the communication between the state machines. The generated code is complemented with run-time libraries implementing the state machine communication.

The generated code is compiled for ARM9 using Gnu C compiler. The compilation includes the codes for eCos, interface functions, external C functions, and the thread configuration. eCos enables the execution and scheduling of multiple threads with multiple priority levels, preemption and timeslicing.

The interface functions are platform dependent, and they connect the generated code with other software components and hardware. The functions take necessary actions for the signals coming out from the application model.

The external C functions are accessed inside the application model to use existing C implementations for algorithms, and to access hardware accelerators. In this case, CRC-8 is implemented as an external C function, and CRC-32 and encryption hardware accelerators are accessed using external C calling functions.

In the hardware implementation, a top-level VHDL code is written to instantiate the hardware components, which have existing VHDL implementations in the library of components. Next, the hardware is synthesized.

A configuration file for Excalibur FPGA is generated by combining the synthesized hardware and executable protocol. Finally, the configuration file is programmed to the flash memory on the development board, and the terminal is ready for use. If the executable protocol is modified, only the configuration file is necessary to regenerate, while the synthesized hardware is retained untouched.

3.3 Execution Monitoring and Model Profiling

The execution statistics include the data throughputs and delays of TUTMAC. The statistics are used for the real-time monitoring of performance, but they can also be stored into a file for the analysis of a larger set of data.

The execution trace contains information about the state transitions, signal transfers, timer events, and thread switching on the protocol. Time stamps for each event are stored. The *UML model profiler* analyses the trace, and combines it with the UML model. The profiler back-annotates the performance information to the UML model. In this way the designer gets feedback to the modeling level.

4 TUTWLAN Terminal Implementation

The final implementation of the TUTWLAN terminal on FPGA is presented in Fig. 6. The processes of TUTMAC are implemented in eCos threads. The processes are divided into two categories, high priority and low priority, each having an own thread with own priority. The high priority thread include processes having real-time constraints, including TDMA scheduling and data processing.

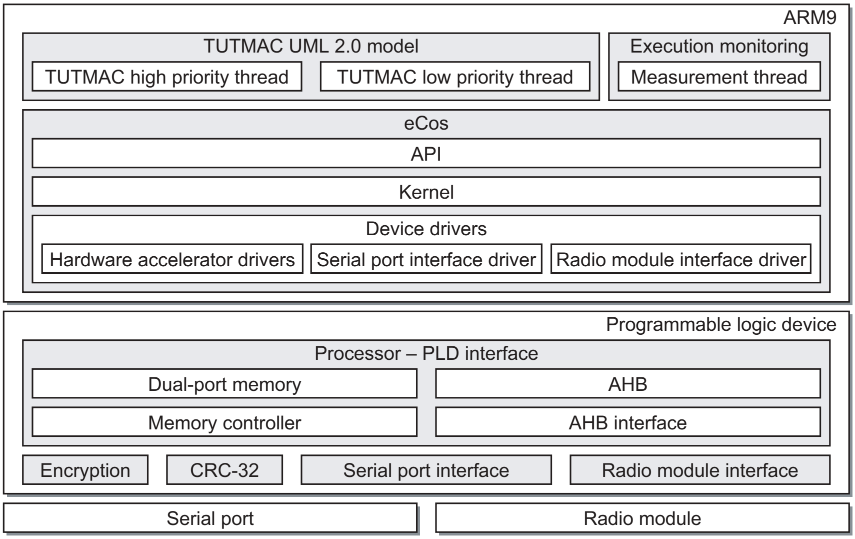


Fig. 6. TUTWLAN terminal implementation on the Excalibur FPGA

The execution statistics for real-time execution monitoring are collected in a measurement thread. The statistics, including data throughputs and delays, are measured continuously, and transferred to a workstation at a second interval.

Custom device drivers and API are included to eCos for accessing the hardware. The device drivers take care of the hardware component control, including transferring data and handling the interrupts. Respectively, API provides common functions and data structures for applications to utilize the hardware components. The hardware accelerators, serial port and radio module interfaces each have a device driver and corresponding API implementations.

Table 1 tabulates the size of the software implementation divided into code and data. In addition to the listed components, 5.6 kB of dynamically reserved memory is required for the general execution, and about 150 kB for data buffering in TUTMAC. The total size of the hardware is 3859 logic cells, which is 92.8 percents of the total PLD capacity of the used FPGA (4160 logic cells).

Table 1. Size of the software implementation

Component name	*Code size*	*Data size*
TUTMAC (generated code)	18.5 kB	2.0 kB
Run-time libraries	14.7 kB	0.8 kB
Interface functions & external C	15.8 kB	1.8 kB
eCos	64.1 kB	7.9 kB
Total	113.1 kB	12.5 kB

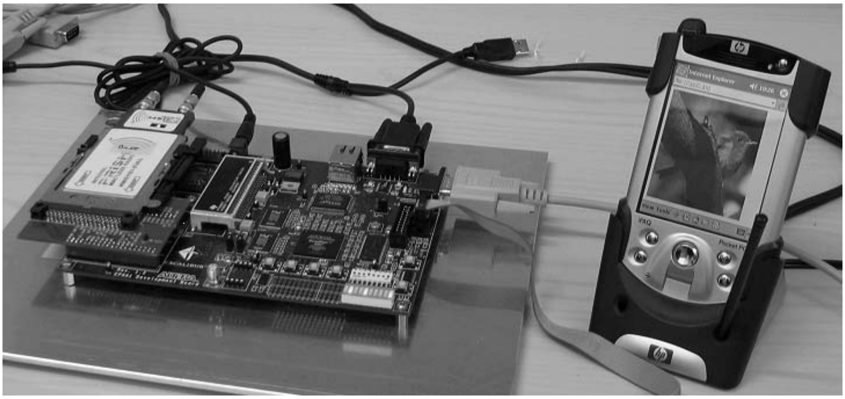


Fig. 7. Prototype containing a TUTWLAN terminal and PocketPC device

4.1 Prototype

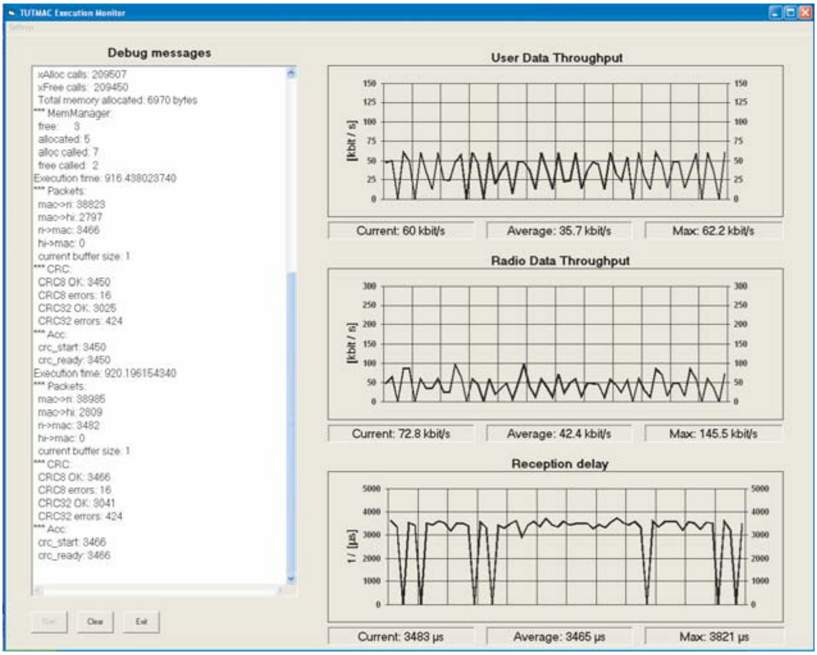
A prototype containing the TUTWLAN terminal and a PocketPC device with an image transfer application is presented in Fig. 7. In all, the prototype contains two TUTWLAN terminals, two PocketPC devices, and a workstation with the execution monitor. The prototype is used to illustrate the real-time execution monitoring and rapid model modifications.

The PocketPC application uses data transfer services of TUTWLAN, and has server and client sides. On the server side, a user selects the images to be transferred. On the client side, the images are received and shown on the screen.

4.2 Execution Monitoring

The execution monitor on a workstation contains three diagrams presenting TUTMAC *user data throughput*, *radio data throughput* and *reception delay* that are measured from the `pUser` and `pPhy` ports showed in Fig. 2(a).

The user data throughput gives the amount of data to the image transfer application. Correspondingly, the radio data throughput gives the gross data transferred between the radio transceiver and TUTMAC. The reception delay is the total execution time for processing a received packet between the radio transceiver and application. The



```
xAlloc calls: 209507
xFree calls: 209450
Total memory allocated: 6970 bytes
*** MemManager:
 free:    3
 allocated: 5
 alloc called: 7
 free called: 2
Execution time: 916.438023740
*** Packets:
 mac->ri: 38823
 mac->hi: 2797
 ri->mac: 3466
 hi->mac: 0
 current buffer size: 1
*** CRC:
 CRC8 OK: 3450
 CRC8 errors: 16
 CRC32 OK: 3025
 CRC32 errors: 424
*** Acc:
 crc_start: 3450
 crc_ready: 3450
Execution time: 920.196154340
*** Packets:
 mac->ri: 38985
 mac->hi: 2809
 ri->mac: 3482
 hi->mac: 0
 current buffer size: 1
*** CRC:
 CRC8 OK: 3466
 CRC8 errors: 16
 CRC32 OK: 3041
 CRC32 errors: 424
*** Acc:
 crc_start: 3466
 crc_ready: 3466
```

Current: 60 kbit/s | Average: 35.7 kbit/s | Max: 62.2 kbit/s

Current: 72.8 kbit/s | Average: 42.4 kbit/s | Max: 145.5 kbit/s

Current: 3483 µs | Average: 3465 µs | Max: 3821 µs

Fig. 8. User interface of the TUTMAC execution monitor

Table 2. Time for each phase of the rapid model modification

Task	*Time*
(UML model modification	30 s)
C code generation	15 s
Compilation	25 s
Flash programming	17 s
Total	(30 s) + 57 s

variation in delay is caused by the varying processing load on TUTMAC as well as the behavior of the application transferring data.

A screenshot of the execution monitor is presented in Fig. 8. The main window contains three diagrams presenting the execution statistics of TUTMAC, and a text window used to output debug messages.

4.3 Rapid Model Modification

The flow enables rapid modification of the functionality and structure of TUTMAC as well as mapping. This gives great possibilities to evaluate different implementations, and compare their performance. The real-time execution monitoring retrieves instant feedback about the modifications. The model profiling back-annotates the performance information to the UML models.

As an example of the rapid model modification, a forward error correction function is added to the UML model. Time for each phase is presented in Table 2. This kind of efficiency can be achieved as the UML models for the application processes already exists, and different combinations are evaluated.

5 Conclusions

This paper presented a new UML 2.0 based design flow that enables the implementation of a real-time embedded system from high-level models. Platform dependent functions are implemented only once, after which new functionality in UML can be created rapidly, and performance constraints verified. The flow was evaluated by implementing the TUTWLAN terminal on a single-processor platform with RTOS and hardware accelerators. A novel feature for performance verification is the back-annotation from the platform.

The ongoing work include multiprocessor implementation with the Koski design flow, and automatic allocation and scheduling of the application model processes on different processors.

References

1. Altera homepage, February 2005. http://www.altera.com.
2. Rong Chen, Marco Sgroi, Luciano Lavagno, Grant Martin, Alberto Sangiovanni-Vincentelli, and Jan Rabaey. *UML for Real: Design of embedded Real-time Systems*, chapter UML and platform-based design, pages 107–126. Kluwer Academic Publishers, May 2003.
3. eCos homepage, February 2005. http://ecos.sourceware.org.
4. Stefania Gnesi, Diego Latella, and Mieke Massink. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *Journal of Logic and Algebraic Programming*, 51(1):43–75, 2002.
5. Panu Hämäläinen, Marko Hännikäinen, Timo D. Hämäläinen, and Jukka Saarinen. Hardware implementation of the improved WEP and RC4 encryption algorithms for wireless terminals. In *Proceedings of the European Signal Processing Conference*, volume 4, pages 2289–2292, September 2000.
6. Marko Hännikäinen, Tommi Lavikko, Petri Kukkala, and Timo D. Hämäläinen. TUTWLAN - QoS supporting wireless network. *Telecommunication Systems - Modelling, Analysis, Design and Management*, 23(3,4):297–333, 2003.
7. Petri Kukkala, Jouni Riihimäki, Marko Hännikäinen, Timo D. Hämäläinen, and Klaus Kronlöf. UML 2.0 profile for embedded system design. In *Proceedings of the Design, Automation and Test in Europe*, volume 2, pages 710–715, March 2005.
8. Luciano Lavagno, Grant Martin, and Bran Selic, editors. *UML for Real: Design of Embedded Real-time Systems*. Kluwer Academic Publishers, May 2003.
9. Object Management Group (OMG). *UML Profile for Schedulability, Performance, and Time Specification (Version 1.1)*, January 2005.
10. Erno Salminen, Vesa Lahtinen, Tero Kangas, Jouni Riihimäki, Kimmo Kuusilinna, and Timo D. Hämäläinen. HIBI v.2 communication network for system-on-chip. In *Proceedings of the International Workshop on Systems, Architectures, Modeling and Simulation*, pages 413–422, July 2004.
11. Telelogic homepage, February 2005. http://www.telelogic.com.

Rapid Implementation and Optimisation of DSP Systems on SoPC Heterogeneous Platforms

J. McAllister, R. Woods, D. Reilly, S. Fischaber, and R. Hasson

ECIT, Queens University Belfast, Northern Ireland Science Park,
Queen's Road, Queen's Island, Belfast, BT3 9DT, UK
{j.mcallister, r.woods, d.reilly, s.fischaber, r.hasson}@ecit.qub.ac.uk

Abstract. The emergence of programmable logic devices as processing platforms for digital signal processing applications poses challenges concerning rapid implementation and high level optimization of algorithms on these platforms. This paper describes Abhainn, a rapid implementation methodology and toolsuite for translating an algorithmic expression of the system to a working implementation on a heterogeneous multiprocessor/field programmable gate array platform, or a standalone system on programmable chip solution. Two particular focuses for Abhainn are the automated but configurable realisation of inter-processor communuication fabrics, and the establishment of novel dedicated hardware component design methodologies allowing algorithm level transformation for system optimization. This paper outlines the approaches employed in both these particular instances.

1 Introduction

The use of digital signal processing (DSP) techniques for reliable data transportation across a given communications channel is becoming increasingly prolific in applications such as mobile communications, radar/sonar systems and image and video compression/decompression. For some of these systems, implementation on embedded multiprocessor systems composed of multiple RISC and DSP software microprocessors has been prevalent and attempts to establish rapid implementation techniques which transform an algorithm level description of the system directly to a working implementation have emerged and matured [1, 2]. Increasingly commonly however, these platforms are complemented, or in some cases replaced by programmable logic devices, in particular field programmable gate array (FPGA), which have evolved to the extent where they can act as system on programmable chip (SoPC) solutions, housing dedicated fast serial communication links, microprocessors, memory and computation units. Exploiting all these resources efficiently using current multiprocessor design techniques is difficult since these have not evolved to a sufficient degree to allow generation of such architectures from a high level algorithm expression.

This paper introduces *Abhainn*, a rapid implementation and exploration methodology and toolsuite under development at ECIT which addresses these problems. Previous works [3, 4] addressing some portions of the methodology are set in a wider context in this paper.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 414–423, 2005.

This paper is organised as follows. Section 2 reviews related work in the area of heterogeneous DSP system implementation and motivates the need for Abhainn. Section 3 introduces the general methodology and system modeling techniques employed in Abhainn. Section 4 briefly outlines the implementation technology, whilst sections 5 and 6 describe dedicated hardware and communication interface synthesis respectively in more detail.

2 Related Work

The relatively recent emergence and rapid evolution of FPGA-based SoPC solutions such as Virtex-II Pro [5] from Xilinx means establishing an effective rapid implementation flow for such a platform is difficult. There are a number of established approaches to rapid implementation on multiprocessor systems and emerging approaches for heterogeneous systems including FPGA, each of which exhibits different characteristics, but a number of important similarities are evident.

The use of dataflow graph (DFG) modelling techniques [6] to model DSP systems and facilitate rapid implementation on multiprocessor systems is a well established in both commercial and academic arenas. Techniques for scheduling and microprocessor software synthesis [2, 7] are mature, and are prompting the diversification of research in this area for increased functional capability. These research efforts have proven fruitful in the emergence of tools for exploitation of the technology, such as GRAPE-II [8], Ptolemy [9], PEACE [10], GEDAE [1] and Compaan [11]. Each of these approaches has a slightly different focus in terms of modeling, implementing and optimizing systems (e.g. GEDAE uses its own dataflow modeling languages and targets multiprocessor systems, whilst Compaan models systems using Kahn Process Networks (KPN) [12] and at present is concentrated on dedicated hardware synthesis for FPGA.) However, there are a number of characteristics common across some or all of these approaches:

1. Implementation independent model of computation (MoC) based specification.
2. Rapid system implementation from algorithm specification.
3. Automated inter-processor communication realisation.
4. Algorithm level transformation for implementation optimization.

These aspects are the key features of any SoPC design methodology. However, current approaches to this problem supply only some of the aspects above. No current solution has comprehensive capabilities for target platform characterization, configurable automated inter-processor communication realization and efficient, configurable dedicated hardware core network synthesis controlled from the algorithm level to trade-off the physical characteristics of the implementation without core re-design, increase core utilization, and balance synthesis between dedicated and programmable resource on FPGA. It is these factors which motivate the need for a methodology such as Abhainn. The general Abhainn system design flow is outlined in section 3.

3 Abhainn Overview

3.1 Overview

The proposed Abhainn system design methodology under development at ECIT is outlined in Fig. 1. There are four major processes: specification (steps 1 and 2), partitioning (step 3), implementation (step 4) and transformation (step 5).

Specification is concerned with defining system behaviour using ideal numerical representations (step 1) and refining this specification to identify the appropriate types of arithmetic and wordsizes required for specific operations (step 2). Dataflow models of computation are a good choice in DSP problem domains due to their semantic similarities with DSP systems, and capabilities for rapid system synthesis. In section 3.2, a new dataflow modeling domain, multidimensional arrayed synchronous dataflow (MASDF) is described which has been developed for Abhainn for DSP system modeling. After specification the algorithm is partitioned amongst the processing resources on the target platform. Portions of an algorithm grouped for mapping to a particular processor (i.e. a microprocessor or FPGA chip) define a partition. This partitioning is external to the Abhainn toolsuite, and the partitioned algorithm is passed to the automated portion for implementation in stage 3.

The Abhainn implementation toolsuite offers an automated transformation route from algorithm level expression to working embedded solution. The partitioned algorithm input is converted to an embedded implementation on a heterogeneous multiprocessor/FPGA or SoPC platform. The designer controls the partitioning of the algorithm, and defines the particular means of communication used at each inter-processor communication point. The operations in this automated portion are outlined in section 4. There are a number of key features:

1. The input partitioned algorithm must be independent of the target platform and the processing and communication technology employed in the implementation. It is the implementation toolsuites job to refine this description to an embedded implementation, as outlined in section 4.
2. The target platform is characterized using a *board support package (BSP)*, which provides platform portability. It simplifies implementation by supplying firmware

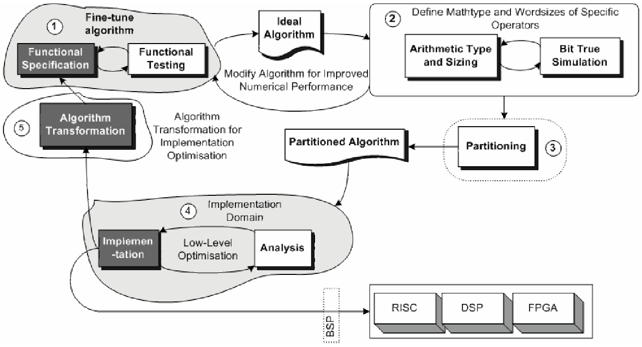


Fig. 1. SoPC Design Methodology Overview

for each platform partition when transforming the partitioned algorithm into an implementation.

3. The toolsuite must be independent of the host algorithm modeling tool.

Of these issues, only the first is addressed in this paper. Post-implementation the application is launched on the embedded platform and analysed for correctness and that real-time constraints are met with algorithm transformation should embedded optimisation be required. The MASDF modeling domain developed to aid this exploration is outlined next.

3.2 Dataflow System Specification

A DFG $G=\{V,E\}$ describes a set of vertices or actors V and a set of edges connecting the actors E describing the dependencies between actors. Actor communication is token based, where a token can be a scalar, vector or matrix to any arbitrary dimension. Actors *fire* by *consuming* tokens through their input ports from the adjoining arcs, performing their specified functionality on the tokens and *producing* the resulting tokens through their output ports onto the adjoining arcs. Every port has an associated *threshold (T)*, which indicates the number of tokens that are consumed/produced at that port in a single firing of the actor. The number of actor firings in an iteration of the schedule is known as its *granularity (G)*.

In DSP applications, and image processing in particular the exploitation of multi-dimensional parallelism to exploit intra-token parallelism is of particular interest [13]. However, the discussion in [4] outlines how relying only on the behavioural semantics of the dataflow domains can hinder existing rapid implementation and exploration methodologies, especially those concerned with architectural synthesis of dedicated pipelined hardware. For this reason, the enhanced structural semantics of multidimensional arrayed synchronous dataflow (MASDF) [4] have been developed, and offer a solution to this issue. This approach is adopted for algorithm modeling in Abhainn.

An MASDF graph $G=\{V_f,E_f\}$ describes a set of actor and arc *families* [14]. The MASDF graph of a matrix multiplication actor is shown in Fig. 2. Actor and arc families are indicated by the presence of shadows. Arcs transport streams of tokens. These are high bandwidth communication channels where data flows throughout the execution of the resulting embedded system. In addition, parameter data is supported, where a parameter is a run-time constant which requires no communication bandwidth after initialization of the embedded manifestation.

The size of the family is denoted above the actors in triangular braces (f_{size} in Fig. 2). Here, alteration of the parameter y by the designer alters the size of the actor family,

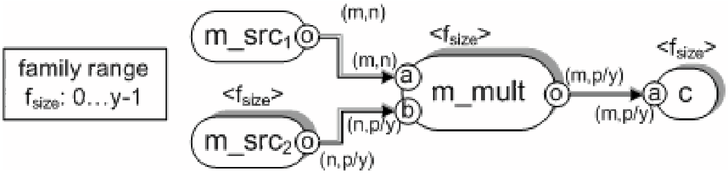


Fig. 2. MASDF Matrix Multiplication

and the size of the matrices (denoted in circular braces in Fig. 2) consumed at port b and produced at port o on each actor instance. In an architectural hardware synthesis methodology for each MASDF actor, if every actor corresponds one-to-one with a parameterised hardware core in the implementation then the designer controls the number of processing components (i.e. resource requirements, throughput) and the size of tokens processed by each (i.e. resource, throughput, latency and memory). This is a powerful graph level optimization capability enabled by the MASDF domain and associated modular hardware synthesis approach.

An MASDF actor is configurable in four aspects: $\mathbf{X}$ the token dimensions at each of the actor ports, G, granularity of the actor in the schedule, $\mathbf{T}$, threshold at each of the actor ports and $\mathbf{S}$, the number of streams impinging on each actor port. It is essential that the cores are flexible for different configurations since this configurability is exploited for system transformation and optimization. Developing rapid, efficient core synthesis approaches which ensure this configurability are ongoing as part of the Abhainn development.

After specifying the system using MASDF, and partitioning the algorithm amongst the platform processing resources, the automated portion of Abhainn, for rapid implementation, is applied. This is described next.

4 Abahinn Implementation Toolsuite Overview

On dividing the algorithm amongst the partitions, inter-processor communication points are inserted where algorithm dependencies cross the boundary between partitions. To implement the partitioned algorithm, three main factors require resolution: software synthesis for software partitions, hardware synthesis for hardware partitions, and generation of a coherent inter-processor communication network from the defined communications points. This process is automated in Abhainn using the tool structure outlined in Fig. 3. There are three major constituent parts to the implementation process: *Linn*, for refinement of software partitions, *Muir* for refinement of hardware partitions, and *Inbhear* for derivation of the inter-processor communication network between them. Each of the three refinement processes occur in parallel in two major stages: *Technology Specific Mapping (TESM)* and *Target Specific Mapping (TASM)*, as outlined in Fig. 3.

The TESM converts the purely algorithmic description of the system behaviour to a description where specific processing and communication cores are inserted in the place of DFG actors to provide a structure to the implementation. Additionally at this stage the inter-processor communication network is inserted and configured as appropriate. The result of the TESM is an implementation architecture to be translated to a working implementation specific to the target devices in the platform. The TASM performs this translation. Note that Inbhear only exists at the TESM level - after technology specific mapping the communications network exists as hardware and software cores which may be integrated and targeted to the specific device with the remainder of the partition in TASM

A great deal of research effort has been expended in academia investigating techniques for software synthesis from dataflow graphs, as employed in Linn [2, 7, 15]. For

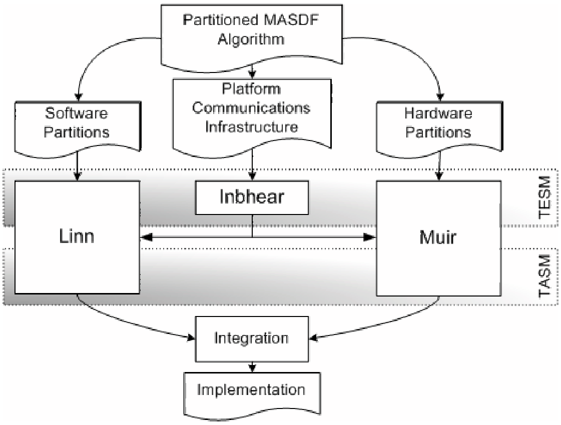


Fig. 3. Abhainn Implementation Toolsuite Architecture

this reason the discussion in the remainder of this section focuses on the other aspects of the Abhainn toolsuite - Muir for dedicated hardware synthesis and Inbhear for communication interface synthesis.

5 Muir - Hardware Partition Implementation

Muir performs the rapid implementation tasks for hardware partitions, and its operation is outlined in Fig. 4. In Abhainn, hardware cores are known as *signal flow objects (SFOs)*. The structure of an SFO is shown in Fig. 5. An SFO is divided into three portions: the *control wrapper*, the *white box component (WBC)*, and the *parameter bank*. The WBC is a core resident in the core library which implements a range of MASDF actor configurations. The technique for generating WBCs is outlined in [3].

In the TESM stage of Muir, this is extracted from the core library and configured via the technique in [4]. Since the SFOs implement cyclic dataflow actors [16], Muir generates the control wrapper for the SFO thats implement the cyclic schedule and switches data into and out of the WBC. After generation of the control wrapper the parameter bank is constructed. As part of this a RAM component is inserted with enough storage for all the parameters, and the address maps for the SFOs generated. These are passed to the host for use during system initialization. During TASM in Muir, the generic SFO structure, described in terms of abstract mathematical components and operations, (e.g. RAMs, delays, etc.) is converted to a target specific description which exploits device specific dedicated computation blocks, storage, and programmable fabric configured for logic, shift registers, distributed RAMs etc.

This synthesis flow can be directed by the designer from the MASDF level as outlined in [4]. Should the core WBC not exist in the library a custom configurable WBC is constructed and inserted in the library for use in further designs.

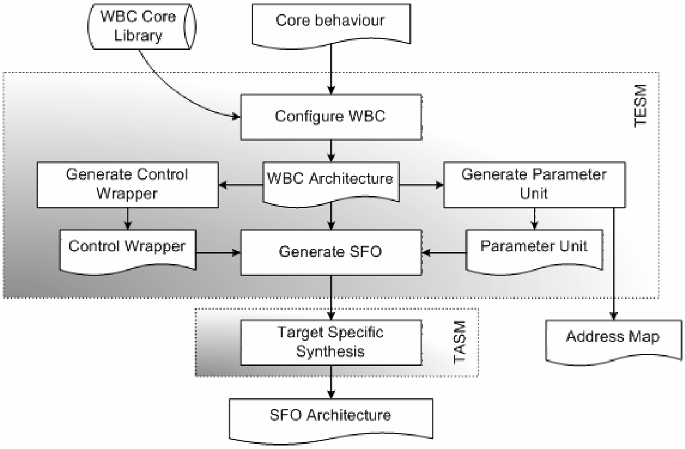


Fig. 4. Muir Methodology

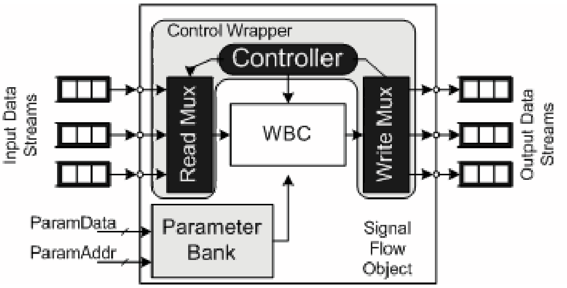


Fig. 5. SFO Structure

6 Inbhear - Communication Interface Synthesis

Inbhear refines the inter-processor communications network to a working implementation. To enable this, the computing platform is considered a distributed computing platform with each node interfacing via complex communication techniques e.g. across a packet-switched network. To support this flexibility, a partition external communications link is visualized as in Fig. 6.

As outlined in section 3, Abhainn must support two types of data traffic: *streams* and *parameters*. Streams are relatively high bandwidth, representing token streams traversing along the vertices of the DFG (e.g. input data samples in a FIR filter). Parameters are written once only, directly into a core on system initialization (e.g. FIR filter tap weights). These represent the two main different types of data traffic used in DSP systems.

A streaming link is described by the designer using the tuple {*physical, VCI*} where VCI is a *virtual channel interface*. As Fig. 6 shows, this defines a three level communications hierarchy. The physical layer connects directly to the pads on the device and

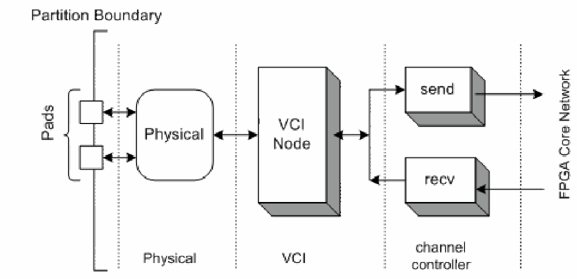


Fig. 6. Abhainn Communications Configuration

retrieves raw binary data from the external network. The VCI acts to multiplex multiple conceptual links onto a single physical link, for instance to implement a packet switch node. Associated with the VCI is a set of *channel controllers* which supply data to, and receive data from the internal functionality of the partition. The Inbhear TESM step proceeds as in Fig. 7.

During *network* level Inbhear processing, the entire communication network which is described in a netlist file known as the *topnet* is analysed, and the communications configurations for each partition extracted and passed on for *partition* level processing. All the necessary information for configuration of the inter-processor communication network is also generated at the network level. For instance, at this level the address maps for all the cores in the FPGA partition are collected, channel numbers assigned for all the data channels (both streaming and parameters), and an initialization function generated for execution on the host during system initialization to write the relevant parameters to the core.

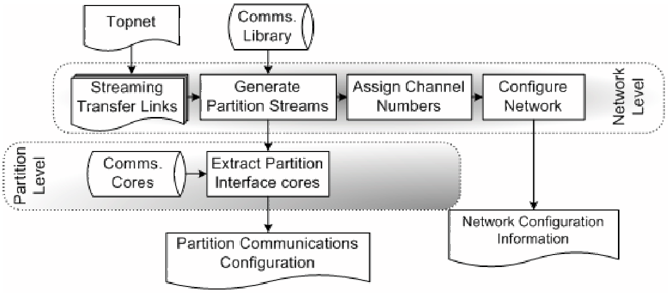


Fig. 7. Inbhear TESM

At the partition level, the technology independent description of the generic stream or parameter link is replaced by a technology specific communication configuration. For instance, in the case where a system where a Myrinet packet switched network operates over a PCI interface, then the configuration {*PCI, Myrinet*} is inserted in place of the generic interface for the partition. In this case the PCI physical and Myrinet VCI are implemented by extracting cores from the comms. core library in Fig. 7 At the partition level, the technology independent description of the generic stream or parameter link is replaced by a technology specific communication configuration.

7 Summary

This paper has introduced Abhainn, a design methodology and automated toolsuite for translating DSP algorithms to a working implementation on multiprocessor/FPGA and SoPC technology. This paper has highlighted the techniques employed to achieve FPGA partition functionality and platform communication infrastructures in a manner which is configurable for designer definition, but also automated in realization. Using a beta-version of the automated toolsuite, for an example adaptive beamforming problem, modelled in GEDAE [1] and targeted to a platform composed of a Pentium-III processor and a Xilinx VirtexII 8000 FPGA communicating via Myrinet over a PCI bus, an unoptimised computation performance of 251 MFLOPs has been achieved.

The described techniques are only two portions of the overall system methodology, which realizes portable multiprocessor/FPGA platform and SoPC system implementation. The Abhainn methodology and toolsuite are evolving and are applied to system domains such as image and video processing (e.g. video conferencing), high end signal processing systems (e.g. radar and sonar), wireless communications systems, particularly multiple-input-multiple-output (MIMO) systems, and physical synthesis systems, for instance electronic synthesis of musical instruments.

Acknowledgments

The contribution of Jasmine Lam and Richard Walke at QinetiQ RTES is acknowledged. This work is supported by the UK MoD Corporate Research Programme, a BAE Systems/EPSRC CASE award, EPSRC grant number C000676/1 and the Department of Eduction and Learning (DEL) NI.

References

1. Madahar, B.K., et. al: How rapid is rapid prototyping? analysis of espadon programme results. EURASIP JASP **2003** (2003) 580–593
2. Sriram, S., Bhattacharyya, S.S.: Embedded Multiprocessors. Marcel Dekker (2001)
3. McAllister, J., Woods, R., Walke, R.: Embedded context aware hardware component generation for dataflow system exploration. In: Samos, Samos, Greece (2004) 254 – 263
4. McAllister, J., Woods, R., Walke, R., Reilly, D.: Synthesis and high level optimisation of multidimensional dataflow actor networks on fpga. In: IEEE Workshop on Signal Processing Systems, Texas, USA (2004) 164 –169
5. Xilinx: Virtex-2 pro platform fpga handbook (2001)
6. Lee, E.A., Parks, T.M.: Dataflow process networks. Proc. IEEE **83** (1995) 773–799
7. Bhattacharyya, S.S.: Software Synthesis from Dataflow Graphs. Kluwer Academic Publishers (1996)

8. Lauwereins, R., Engels, M., Ade, M., Peperstraete, J.: Grape-2: A rapid prototyping environment for dsp applications. Computer **28** (1995) 35–43
9. Hylands, C., et al.: Overview of the ptolemy project. Technical Memorandum UCB/ERL M03/25, University of California at Berkeley (2003)
10. Laboratory, C.A.P.: PEACE Users Manual v.1.0b. Seoul National University. (2004)
11. Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., Deprettere, E.: System design using kahn process networks: The compaan/laura approach. In: Design Automation and Test in Europe. Volume 1., Paris, France (2004) 340 – 345
12. Kahn, G.: The semantics of a simple language for parallel programming. In: Proc. IFIP Congress, North Holland Publishing Company. (1974)
13. Murthy, P.K., Lee, E.A.: Multidimensional synchronous dataflow. IEEE Trans. Signal Processing **50** (2002) 2064–2079
14. Kaplan, D.J., Stevens, R.S.: Processing graph method 2.1 semantics. (2002)
15. Park, C., Jung, J., Ha, S.: Extended synchronous dataflow for efficient dsp system prototyping. Design Automation for Embedded Systems **6** (2001) 295–322
16. Bilsen, G., Engels, M., Lauwereins, R., Peperstaete, J.: Cyclo-static dataflow. IEEE Trans. Signal Processing **44** (1996) 397–408

DVB-DSNG Modem High Level Synthesis in an Optimized Latency Insensitive System Context

P. Bomel[1], N. Abdelli[2], E. Martin[1], A.-M. Fouilliart[2], E. Boutillon[1], and P. Kajfasz[2]

[1] LESTER Laboratory, CNRS FRE2734, UBS University,
56321 Lorient Cedex, France
surname.name@univ-ubs.fr

[2] THALES Communications, 160, boulevard de Valmy BP 82,
F92704 Colombes Cedex, France
surname.name@fr.thalesgroup.com

Abstract. This paper presents our contribution in terms of synchronization processor to a SoC design methodology based on the theory of the latency insensitive systems (LIS) of Carloni et al.. This methodology 1) promotes pre-developed IPs intensive reuse, 2) segments inter-IPs interconnects with relay stations to break critical paths and 3) brings robustness to data stream irregularities to IPs by encapsulation into a synchronization wrapper. Our contribution consists in IP encapsulation into a new wrapper model containing a synchronization processor which speed and area are optimized and synthetizability guarantied. The main benefit of our approach is to preserve the local IP performances when encapsulating them. This approach is part of the RNRT ALIPTA project which targets design automation of intensive digital signal processing systems with GAUT [1], a high-level synthesis tool.

1 Introduction

Modern integrated circuits (ICs), named "systems on a chip" (SoCs), are the composition of several sub-systems exchanging data. SoC size increase is such that an efficient and reliable interconnection strategy is now necessary to combine sub-systems and preserve, at an acceptable design cost, the speed performances provided by a very deep sub-micron technologies [2]. This communication constraint can be satisfied by a rapid enough communication media between IPs. Because they are easy to deploy and area efficient, bus topologies are frequently used.. Nevertheless they introduce a bottleneck as 1) only an IP can send a data at a time and 2) wires length and capacitive load of the numerous connected IPs increase the propagation time and reduce the maximum frequency. Hierarchical and scalable bus topologies [3] contribute to push back these limits by segmentation of the bus into several buses linked through bridges. Thus better communication parallelism and wire length reduction are obtained. High performances SoCs now need micro-networks architectures, named "networks on chip" (NoC [4]), inside which the parallelism degree and the data throughput are adapted to the application needs. NoCs integrate supplementary services like packet routing, data flow control and eventually error detection. These services have a non negligible area cost. To overcome

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 424–433, 2005.

the propagation delay increase along interconnects in single-clock synchronous ICs, the ”globally asynchronous locally synchronous” (GALS [5]) system approach divides the whole system into synchronous and clock-independent sub-systems communicating through point to point asynchronous channels. The interface between sub-systems and network is a decisive design issue for the applicability of the methodology because it must be reliable and must not impact the local sub-systems frequencies. A very promising evolution of GALS methodology is the theory of the ”latency insensitive systems” (LIS [7]) which communication network is synchronous. It shares with NoCs the high degree of potential parallelism and with the buses the deployment ease.

This paper is organized as follows. First, section 2 provides background info on the synchronization of IPs within the communication network in the ”latency insensitive systems” methodology context. Section 3 experimentally justifies our approach, specifies our synchronization processor model, and provides comparative results of synthesis for FPGA against classical finite state machines (FSM). Section 4 presents a design experience of a DVB-DSNG signal processing chain in the context of the high-level synthesis tool GAUT. A frequency/phase/frame synchronization and a 64 states Viterbi decoder IPs are compared to their hand-coded RTL equivalent implementations in Section 5. Section 6 concludes this paper and gives an overview of future research extensions.

2 Related Works

GALS systems, introduced by [5], combine synchronous sub-systems communicating through an asynchronous network and require the design of specific interfaces [6]. They also rely on the possibility to stop at will any synchronous block and make use of gated clocks to synchronize sub-systems computing with data flow irregularities and save energy while in a waiting state. A way to avoid the design of a mixed synchronous/asynchronous IC is to apply the theory of the ”latency insensitive systems” [7] which communication network is not asynchronous but pseudo-asynchronous. This network is build from pipelined point to point channels where pipelining is implemented by insertion of intermediate storage cells named relay stations. The objective is to store and forward data flowing from one IP to another and support the communication protocol between IPs and relay stations. Transit latencies between IPs are no more unpredictable but become multiples of the communication network clock period. LIS theory allows then to reuse commercial RTL synthesis tools as well as for the IPs than for the pseudo-asynchronous network. Asynchronous components insertion is not necessary any more. This theory also formally proves [8] the ”latency-equivalence” between the pure synchronous system model and the pseudo-asynchronous one. Patient processes [9] are a key element in the LIS theory. These suspendable synchronous IPs (named pearls) are encapsulated into a wrapper (named shell) which function is to make them insensible to the IO latency and to drive the IP’s clock. The decision to drive or not the IP’s clock is implemented in a very efficient way using combinatorial only. The LIS methodology is then divided in two steps. The first step consists in the encapsulation of all suspendable IPs into synchronization wrappers before an initial physical synthesis (place&route).

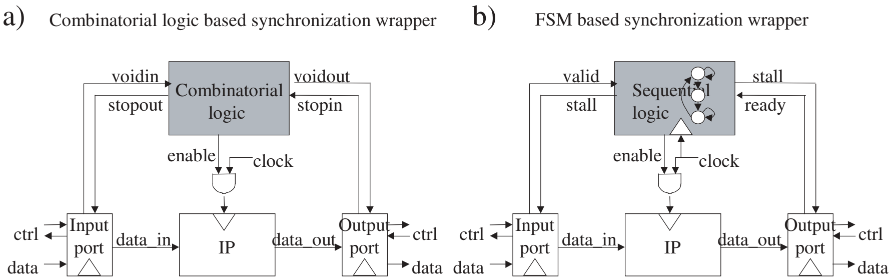


Fig. 1. (a) Carloni et al.'s patient process model and (b) Singh and Theobald's model

Second step consists in the identification of critical paths on the communication network and insertion of relay stations. The network becomes pseudo-asynchronous and can be placed and routed again. The relay stations insertion, which aims at breaking the critical paths, can induce a severe system level performances degradation if there are feedback paths. [10] formally specifies this phenomenon so as to quantify the relay stations insertion impact and then to drive it. Nevertheless, the LIS approach relies on a simplifying, but restricting, assumption: an IP is activated only if all its inputs are valid and all its outputs are able to store a result produced at next clock cycle. Now, it is frequent that only a subset of the inputs and outputs are necessary to execute one step of computation in a synchronous IP. To limit the patient process sensitivity to the only relevant inputs and outputs, Singh and Theobald [11] suggest to replace the combinatorial logic that drives the clock by a Mealy type FSM. This FSM tests the state of only the relevant inputs and outputs at each cycle when data are expected. We call these cycles "wait points" from here. This approach is an extension of the LIS one and has the advantage to correspond to a more realistic communication behavior. It can be implemented if one disposes of input and output schedules that prove the IP communication behavior is cyclic and not data-dependent : i.e. it is statically predictible.

Finally, in order to reduce hardware cost, Casu and Macchiarulo [12] prove that, if it is possible to determine a static scheduling of all the IPs activation, then the relay sta-

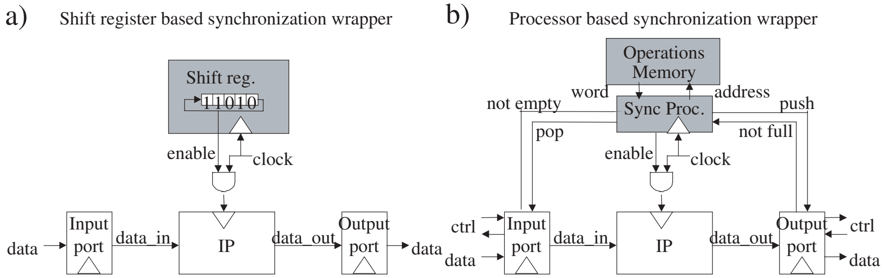


Fig. 2. (a) Cassu and Macchiarulo's model and (b) **New approach** patient process model with a synchronization processor

tions can be replaced by simple flip-flops and the synchronization upstream and downstream protocol signals can be definitely removed. The IP activation static schedule is implemented with a shift register which content drives the IP's clock. This approach relies on the hypothesis that there are no irregularities in the data streams: it is never necessary to randomly freeze the IPs. This approach is applicable only to systems which environment produces data and consumes results faster than the systems can compute.

3 New Approach – A Smaller Wrapper

As 1) LIS methodology lacks the ability to dynamically sense IO subsets, 2) FSMs can become very large as communication bandwidth does and 3) shift register based synchronization targets only extremely rapid environments, we propose to encapsulate IPs into a new synchronization wrapper model which area is much less than the FSM-based wrappers area, speed is enhanced (mostly thanks to area reduction) and synthesizability is guaranteed whatever the communication schedule is.

3.1 Motivation

Singh's approach relies on a FSM model to limit the patient process sensitivity only to relevant inputs and outputs. This FSM complexity depends on the total number of cycles an IP needs for a given communication schedule and on the number of inputs and outputs surrounding the IP. Our experiences in digital signal processing systems prototyping on FPGAs demonstrated us that FSM synthesizability is a delicate issue. Available synthesis tools have a rather high sensitivity to FSM length (number of states) and width (number of input and output signals) and might not synthesize all FSMs. To justify this, we have made synthesizability tests with various representative FSMs. By representative, we mean "as complex as" the communication schedule for some digital signal processing well known algorithms.

We have chosen the Decimated In Time (DIT) radix-2 Discrete Fourier Transform (DFT) and the Discrete Cosine Transformation (DCT) as valuable references to define a communication schedule complexity reference. With the help of the GAUT [13] high-level synthesis tool we have produced several RTL implementations for a common target: a Xilinx VirtexE-1000 clocked at 100 MHz. An extract of the test campaign is presented in Table 1. Columns contents are (from left to right): algorithm size order in points, samples arrival cadency, sum of input and output ports, number of wait points (data synchronization), and average number of computing cycles (no data synchronization) between two wait points. These measures show, in a particular but real context, that a communication schedule contains numerous wait points and that the number of ports raises as the cadency is reduced. A few tenth of ports and a few hundredth of wait points are conceivable.

Our FSMs depends on three parameters: number of ports, number of wait points and number of average cycles between wait points. Its pseudo-VHDL model is described by Figure 3. This FSM model 1) pilots the IP's clock with the *enable* signal, 2) activates only the relevant intermediate storage units of the pseudo-asynchronous network (working like a set of FIFOs) with the generic signals *pop_push_fifos*, and 3) *check_data* and *check_room* are generic expressions that respectively check if data are valid on relevant

Table 1. FFT & DCT high-level synthesis results

	Points	Cadency (μs)	I/O Ports	Wait points	Run cycles
DIT	128	8,4	16	197	43
		5,1	25	156	33
	256	38,4	6	515	85
		19,2	9	381	50
	512	87,1	6	1027	85
		43,8	9	765	57
DCT	4	1,0	4	39	26
		0,5	8	19	26
	8	3,4	9	91	37
		2,0	16	69	29

```
process (rst, clk)
begin
   if rst = '1' then
      enable <= '0' ;
      <pop_push_fifos> <= '0' ;
      s := S0 ;
   elsif clk = '1' and clk'event then
      case s is
      when <a synchronization state> =>
         if <check_data> and <check_room> then
            state := <nextstate> ;
            enable <= '1' ;
            <pop_push_fifos> <= '1' ;
         else
            enable <= '0' ;
            <pop_push_fifos> <= '0' ;
         end if ;
      when <a computing state> =>
         state := <nextstate> ;
         <pop_push_fifos> <= '0' ;
      end case ;
   end if ;
end process ;
```

Fig. 3. SP pseudo-VHDL specification

input ports and if room is available on relevant output ports. A brief extract of the FSM synthesis is given in Table 2. Column contents are (from left to right): number of ports, number of wait points, average number of computing cycles between two synchronization points, area occupied in slices and maximum obtainable frequency. Then, we have generated and synthesized these representative VHDL FSMs with Xilinx's tool XST.

These synthesis results illustrate that the area increase is not linear, raises with the schedule complexity and unfortunately reaches a ”red brick wall” in the representative

Table 2. FSM physical synthesis results

Ports	Wait points	Run cycles	Size (slices)	Max freq (MHz)
4	4	2	15	162
8	8	4	36	148
16	16	8	110	123
32	32	16	429	105
64	64	32	Impossible.	Impossible.

FSMs space. Our goal is absolutely not to criticize any synthesis tools but rather to simply state they have limits we must live with. However, this extremely restrictive fact for our experiences motivated us to search for a new synchronization wrapper architecture which would be synthesizable with (almost) all RTL and physical synthesis tools. To end this section, we also had to verify this situation was not due exclusively to ISE. So we replayed it with Quartus and DC (semiconductor technologies ranging from .35 μm to .15 μm), got the same results and came to the same conclusion.

3.2 Implementation and Results

Our solution is equivalent to Singh's FSMs. This is a controller that reads and executes cyclically operations stored in a memory. We name it a *synchronization processor* (SP). Figure 2b specifies the new synchronization wrapper structure with our SP. The SP communicates with the ports through FIFO-like signals. These signals are equivalent to the *voidin/out* and *stopin/out* of [7] and *valid*, *ready* and *stall* of [11]. Number of input and output ports can be any. The clock is driven by the *enable* signal. The SP model is specified by a three state FSM depicted by Figure 4a. Its data path is composed of: a program counter *pc* and a run cycles counter *cpt*. Generic expressions *I(pc)*, *O(pc)* and *N(pc)* represent the decoding of the operation word and allows us to test various coding formats with little changes in the SP. They respectively represent the relevant input ports, output ports masks and the number of run cycles. The SP is in the *reset* state at power up and switches between the two *wait* and *run* states depending on whether if it is blocked on a waiting point or if it is freely computing without synchronization with data. To avoid unnecessary signals and save area, the operation memory is an asynchronous ROM (or 4Kbits SelectRAM DPRAMs for FPGAs)

and its interface with the SP is reduced to two buses : the *operation address* and *operation word*. In all tested cases, no more than 2 SelectRAM DPRAMs have been necessary to store the operation memory. Therefore we neglected this memory area in the SP area reports. The VHDL specification of a three states FSM is obvious and we give now (Figure 4b) a curve which situates the synthesis results for a VirtexE-1000. FSM*X*X* notation means *FSMports*X*waitstates*X*runcycles*. The selected SP has 16 input and 16 output ports, its area is 16 slices large and it maximum frequency is 124 MHz. Compared to an equivalent FSM16x16x16, it has the same frequency but saves up to 95 % of slice area (and consumes only one SelectRam bloc for the operation memory). Finally, its frequency does not depend on the total number of cycles it needs to accomplish a full computation. It is then more rapid than any longer (in cycles) FSM.

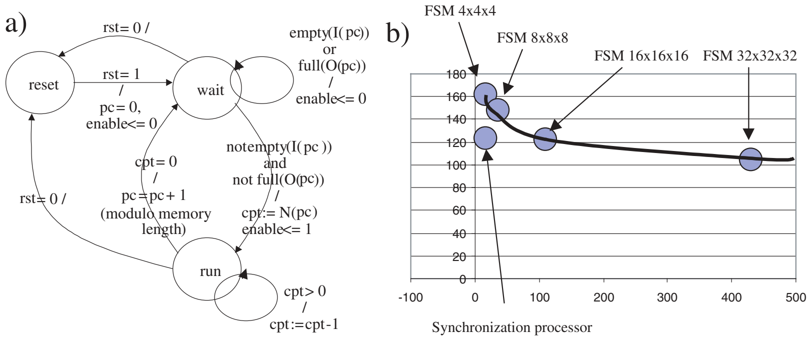


Fig. 4. (a) SP specification and (b) SP vs FSM area and speed

Conclusion. *Our SP has an essential characteristic: its complexity does not depend on the number of cycles the IP needs for a whole computation but only on the number of ports and maximum number of run-cycles.*

4 DVB-DSNG Experience

This paragraph depicts an experimentation of the synchronization wrapper model for LIS patient processes with the synchronization IP cores and Viterbi IP cores implemented in a complete receiver compliant with the DVB-DSNG standard [14]. Digital Satellite News Gathering (DSNG) and Digital Video Broadcasting applications by Satellite (DVB_S) consist in point-to-point or point-to-multipoint transmissions, connecting fixed or transportable up-link terminal and receiving stations, not intended to be received by the general public. This standard [14] allows transmissions from 1.5 Mbps to 72 Mbps. The Figure 5 presents the main elements of a receiver compliant with DVB-DSNG standard. It's composed of:

- A demodulation module of the signal associated to the DVB.
- An IP Viterbi (7,1/2) block.
- An IP Reed Solomon (204,188) block.

In current digital communication systems based on iterative detection/decoding, it is necessary to determine the likelihood that a particular symbol has been transmitted.

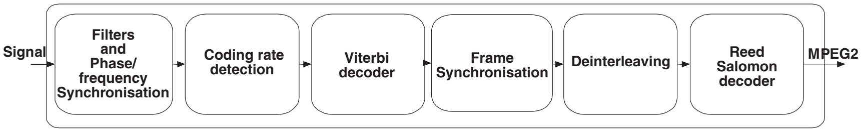


Fig. 5. Receiver scheme compliant with DVBDSNG standard

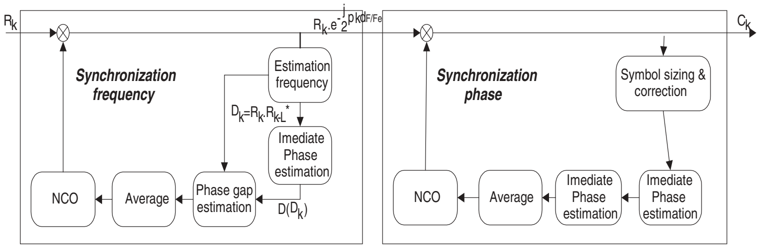


Fig. 6. Synchronization IP architecture

The Viterbi algorithm is based on a symbol-wise maximum a posteriori probability criterion and proved its performance for estimating the states or output of a Markov chain observed in white noise. A Viterbi decoder can be divided into three synchronous circuit blocks: branch metric (BM) unit, add compare and select (ACS) unit and survivor memory evaluation (SME) unit. Each time slot, the BM unit determines the distance between the received symbol and a noiseless symbol. The performance of a Viterbi decoder is actually limited by the ACS unit: for each of the states, the current state metric has to be known before the next state metric can be calculated. For such high data rates, the REA technique is used [15]. This method requires a shift register, which contains the survivor path leading to this state. The registers are trellis-like interconnected and their update is performed with an exchange of their contents based on the new decisions provided by the ACS unit. The DVB-DSNG decoder synchronizes the video frame that it receives in phase and frequency with the synchronization IP. The *synchronization* IP can be divided into two synchronous circuit blocks:

- An algorithm of phase synchronization to collect the phase gap as well as the weak variations of phase.
- An algorithm of frequency synchronization to get the frequency gap to 6 % of the symbol output.

All *Synchronization* IP are composed by an estimation function of the ratio signal/noise and a function that estimate the residual phase. These functions were added to use the knowledge of the decoded symbols.

5 Comparative Results

The LIS methodology has been tested in the DVB-DSNG IP cores. To check its validity, we have synthesized the synchronization and the Viterbi IPs with GAUT and measured the obtained frequencies under various cadency constraints. We have also compared the generated codes with their hand-coded RTL equivalent. We have tested the synchronization IP with a Virtex-E device. Figure 7a compares two curves. The square points one represents the various frequencies obtained with a hand-coded synchronization IP. The diamond-shaped points one represents the frequencies obtained when synthesizing the algorithmic IP with GAUT. It shows that 1) generated RTL codes behave closely to

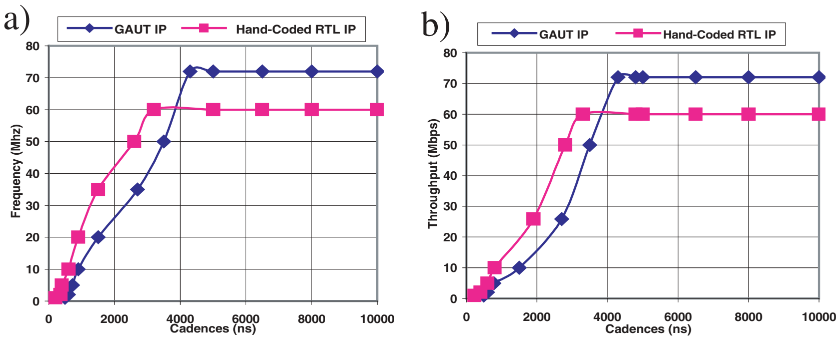


Fig. 7. (a) Synchronization IP and (b) Viterbi IP

hand-written RTL codes in term of frequencies and 2) a better maximum frequency can be automatically reached: i.e. a faster 75 MHz architecture has been obtained by synthesis and must be compared with the maximum 65 MHz frequency of a hand-coded RTL IP.

We have tested the Viterbi IP with a Virtex-400 to validate the results with more than a single family target. We can see in Figure 7b the results of the Viterbi decoder IP synthesis. We can see in particular that the *synchronization processor* of the Viterbi decoder IP still preserves the maximum throughput rate. We can also note that the throughput obtained with RTL level architecture of GAUT synthesis tool (72 Mbps) is again greater than the RTL level architecture value (60 Mbps).

6 Conclusion and Future Work

This paper presents our *synchronization wrapper* for LIS patient processes that better preserves maximum frequency at IP level. Comparative results of physical synthesis for the SP against FSMs prove our SP can provide important gains in area and speed. An industrial synthesis experiment of a synchronization and Viterbi IPs for a DVB-DSNG modem illustrates that 1) obtained frequencies are in line with the ones of hand-coded RTL code and 2) better frequencies are reachable when relaxing the time constraint. Several research axis to this work are under consideration: area optimization and concept validation with an ASIC technology (0.15 μm technology from STMicroelectronics) and an extension of the SP to data-dependant scenarios. Data-dependant scenarios need a new type of operations: i.e. test and branching ones. Then our SP will get closer to a regular processor with an instruction set rather than to a simple controller.

References

1. GAUT. web site, http://web.univ-ubs.fr/gaut
2. International Technology Roadmap for Semiconductors. (2003)

3. Dawson, W.K., Dobinson, R.W.: Buses and bus standards. Computer Standards & Interfaces **20** (1999) 210–224
4. Benini, L., De Micheli, G.: Networks on Chips: A New SoC Paradigm. IEEE Computer (2002) 70–78
5. Chapiro, D.M.: Globally-Asynchronous Locally-Synchronous Systems. PhD Thesis, Stanford University (1984)
6. Chakraborty, A., Greenstreet, M.R.: Efficient self-timed interfaces for crossing clock domains. In: Proceedings of the International Symposium on Asynchronous Circuits ans Systems (ASYNC'03), Vancouver (2003)
7. Carloni, L.P., McMillan, K.L., Sangiovanni-Vincentelli, A.L.: Theory of Latency-Insensitive Design. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems **20** (2001)
8. Carloni, L.P., Sangiovanni-Vincentelli, A.L.: A Formal Modeling Framework for Deploying Synchronous Designs on Distributed Architectures. In: First International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous (FMGALS'03), Pisa (2003).
9. Carloni, L.P, Sangiovanni-Vincentelli, A.L.: Coping with Latency in SoC Design. IEEE Micro, Special Issue on Systems on Chip **22** (2002) 24–35
10. Carloni, L.P, Sangiovanni-Vincentelli, A.L.: Performance Analysis and Optimization of Latency Insensitive Systems. In: Proceedings of the 37th Design Automation Conference (DAC'00), (2000)
11. Singh, M., Theobald, M.: Generalized Latency-Insensitive Systems for Single-Clock and Multi-Clock Architectures. In: Proceedings of the Design Automation and Test in Europe Conference (DATE'04), Paris (2004)
12. Casu, M.R., Macchiarulo, L.: A New Approach to Latency Insensitive Design. In: Proceedings. of the Design and Automation Conference (DAC'04), San Diego (2004)
13. Corre, G., Senn, E., Bomel, P., Julien, N., Martin, E.: Memory Accesses Management During High Level Synthesis. In: Proceedings of the Intl. Conf. on Hardware/Software Codesign and System Synthesis (CODES'04), Stockholm (2004)
14. Standard ETSI EN 301 210, Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for Digital Satellite News Gathering (DSNG) and other contribution applications by satellite. (1999)
15. Biver, M., Kaeslin, H., Tommasini, C.: Architectural design and realization of a single-chip Viterbi decoder. Integration, the VLSI Journal **8** (1989) 3–16

SystemQ: A Queuing-Based Approach to Architecture Performance Evaluation with SystemC

Sören Sonntag[1], Matthias Gries[2], and Christian Sauer[2]

[1] Infineon Technologies, Wireline Communications, Munich, Germany
[2] Infineon Technologies, Corporate Research, Munich, Germany
{Soeren.Sonntag, Matthias.Gries, Christian.Sauer}@infineon.com

Abstract. Platform architectures for modern embedded systems are increasingly heterogeneous and parallel. Early design decisions, such as the allocation of hardware resources and the partitioning of functionality onto architecture building blocks, become even more complex and important for the resulting design quality. To effectively support designers during the concept phase we base our design flow SystemQ on queuing systems. We show how by starting with a performance model the system's behavior and structure can be refined systematically. SystemQ is implemented in SystemC and seamlessly supports the refinement of SystemQ models down to established transaction and RT levels. Compared with existing approaches, SystemQ's formalism exposes transaction scheduling as one key aspect of the system's performance and allows the modeling of time and resource workload-dependent behavior. A case study underpins the usefulness of SystemQ's approach by evaluating a network access platform at three refinement levels.

1 Introduction

Embedded systems are facing new design challenges due to recent trends in applications and technology. In our experience the computational complexity of algorithms needed for, e. g., multi-standard wireless terminals and network access concentrators grows faster than the hardware integration complexity dictated by Moore's law. In addition, due to stringent constraints on platform costs and power dissipation, novel parallel and programmable application-specific architectures are increasingly being deployed. The designer is now faced with coordinating the execution of the application on several processing elements and accelerators. This means that early design decisions on mapping functionality onto computing resources, scheduling of processing elements and among building blocks, and data placement can significantly affect the quality of the final design.

Last but not least, newly arising application domains reveal properties that have not been considered in the past. A prominent example is network processing. Superficially, traffic flows resemble streams similar to media processing applications. However, network protocols and processing tasks reveal time and workload-dependent behavior unknown from other data flow dominant domains. Data streams (network traffic) must be distinguishable with respect to origin, destination, and data type to meet Quality-of-

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 434–444, 2005.

Service (QoS) requirements by, for instance, modifying the processing order on hardware resources. If a packet is queued for a long period because of backlog at a congested resource, a protocol may decide to drop the packet, e. g., see voice traffic with tight latency constraints. A final example is dynamic load balancing of non-deterministic traffic.

From these trends we can derive the following requirements on a design flow for modern platform architectures:

- Performance evaluation must already be supported during the concept phase to guide design decisions at the system-level, where these decisions have the highest impact on the resulting design quality.
- Many alternatives of allocating resources and mapping functionality onto building blocks of the platform must be reviewed during the concept phase. The framework must be considerably faster than established RTL and transaction level evaluation.
- During concept phase, RTL or transaction level representations of the architecture are either not available or too complex for evaluating many alternatives. Representations are needed that allow us to express and explore the features we are interested in (time dependent behavior, scheduling, resource mapping) while providing us with precise quantitative performance results.
- Aspects of the platform architecture should be separated from the description of the system functionality to enable an unbiased assessment of design alternatives and to encourage reuse of models. This concept is also known as the separation of concerns principle in literature [1].
- The refinement of application and architecture models down to the actual implementation of the system must be supported to establish a systematic and reliable approach, where immediate results from verification and evaluation can be used to constrain further refinement levels.

Based on these assumptions we present a fresh approach, called SystemQ, to platform architecture evaluation and refinement targeted at early design phases of system development. Our contributions can be summarized as follows:

- Our modeling approach is based on queuing theory where scheduling and workload dependencies are made explicit by the semantics of the formalism.
- Timing dependencies between functional and architectural aspects (due to the processing capabilities and workload of a hardware resource) can be represented since timing information is fed back into the model, possibly affecting the functionality of the system at run-time.
- In order to stimulate wide acceptance among platform designers our framework is based on SystemC. Existing code bases for algorithms in C and C++ can be reused in our models.
- We provide a systematic path of refinement steps starting from plain performance models down to transaction-level and RTL exploiting SystemC's refinement methodology.

The rest of the paper is structured as follows. In the next section we briefly introduce the theory of queuing systems. Section 3 continues with a description of how queuing systems can be implemented using SystemC. In Section 4 we introduce our design methodology and levels of abstraction. A case study of evaluating a network access platform using our approach is shown in Section 5. Related work is discussed in Section 6. We conclude our paper in Section 7.

2 Queuing Systems

Many formalisms exist for modeling control and data flow. Petri nets, for example, provide powerful abstractions that are analyzable in many ways. Kahn process networks on the other hand are often used in practice for data flow applications and signal processing systems [2]. Control flow is often modeled using finite state machines and variants thereof. Besides control and data flow modeling we are specifically interested in certain performance relevant aspects of control flow, namely scheduling and workload-dependent behavior. Thus, we choose queuing systems since they provide explicit scheduling inherently. A brief introduction to queuing theory is provided next. A detailed description can be found, e. g., in [3].

Queuing systems consist of *queues* and *servers*, see Figure 1. A queue is a waiting room where transactions (also called requests or customers) are stored until a server becomes available to process the transaction. The processing takes a specific amount of service time after which the transaction leaves the queuing system. Using queuing systems we can determine, e. g., the residence time of transactions in the system, the average queue length, and the server utilization.

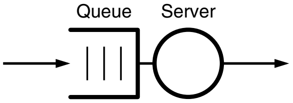


Fig. 1. A queuing system consisting of queue and server

Queues store transactions according to their queuing discipline. Common disciplines are, e. g. first-in-first-out (FIFO), last-in-first-out (LIFO), or generalized processor sharing. Thus, queues are used for explicit scheduling of transactions, but they neither modify them nor consume time. Often, queues are unbounded, i. e. their storing capacity is infinite.

Servers consume an amount of time while processing transactions sequentially, called *service time*. The service time may be constant for all transactions, may be a time distribution, or may depend on the transaction class. In systems that use distinct transaction classes [4] servers can also alter the class of transactions while processing them. Furthermore, servers may create or delete transactions but do not store them.

For more realistic models several queuing systems are connected with each other [3, 4]. Such *networks of queuing systems* can be solved analytically. This, however, is a cumbersome task, especially for larger systems, non-exponential arrival time distributions, and transaction-dependent service times as required in our case. Thus, we use simulation for evaluation.

3 SystemC Based Simulation

For our SystemQ simulation environment we rely on SystemC (*www.systemc.org*) since it has become a wide-spread open-source modeling language for embedded systems spawning design and verification on different abstraction levels from concept to implementation in hardware and software [5]. We exploit SystemC's 2.0 communication support and its discrete event simulation capabilities for our timed queuing models by providing component and communication libraries as explained in this section. This way performance information, such as latency or throughput of the overall system or individual building blocks, can be derived and potential bottlenecks can be identified. We base subsequent architectural exploration, refinement, and implementation steps, described in the next section, on SystemC's methodology.

3.1 Queuing Models in SystemC

Figure 2 shows a queuing system in SystemC. Both types, queues and servers are *sc_modules* that can be executed concurrently by using *processes*. Networks are composed from these elements by connecting ports with directed point-to-point *sc_channels*. Queues, modeled as *hierarchical channels*, are directly connected to servers. This way queuing network semantics are enforced automatically at design time (Fig. 2b). Our SystemQ library follows a structured implementation style, as shown in Figure 3. Elements are derived from abstract parents that provide default functionality. Particular functions, e. g. queuing disciplines in case of queues, are implemented in separate library elements to increase re-use. Elements may be implemented in different abstractions to enable refinement steps, e. g. the untimed, delayed, and rated channels.

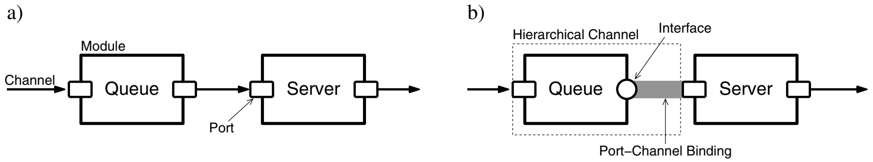


Fig. 2. Queuing system construction in SystemC: a) Based on *sc_channels* and *sc_modules*, and b) using hierarchical channels to enforce queuing model semantics

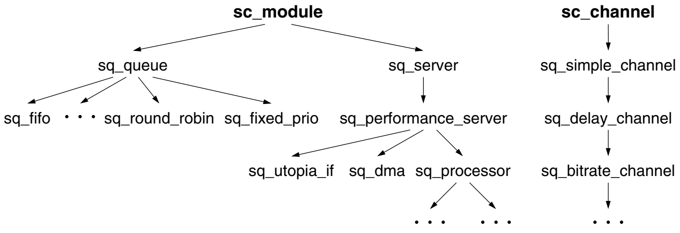


Fig. 3. SystemQ library structure with elements for our case study in Section 5

3.2 Supporting Framework

Apart the rich class library our SystemQ environment provides support for (re-)configuration, stimulation, and verification of queuing models.

All classes provide, for instance, an interface for their file based configuration at run-time. This enables faster turn around cycles in case of parameter explorations without recompilation. The framework furthermore includes modular and extensible means for generating realistic traffic loads on different abstraction levels. A number of elements can be composed to form layered sources, e. g. packet generators. Following the OSI/ISO approach such sources first generate a chunk of data based on a particular distribution (time, length) which then is encapsulated/transformed repeatedly depending on the desired protocol stack. Similarly, layered sinks can be employed.

4 Performance Modeling and Refinement

The main focus of queuing systems is on statistics, e. g. to determine queue lengths, server utilization, and residence times of transactions in the system. These properties are inter-dependent and are effected by transaction arrival processes, transaction classes, and queuing disciplines. With SystemQ's simulation approach we gain additional information since we can trace time dependencies, profile individual resources, and follow the processing of particular transactions through the model.

The quality of results depends on the abstraction level. Initially, a rough estimation of the overall system performance is required. Refinement enables better results by adding details either in function, structure, or communication. SystemQ applies SystemC's refinement strategies to queuing networks.

4.1 Functional Refinement

Functional refinement may add (and refine) the function of servers and/or introduce distinguishable transaction classes.

On the performance level a server has only three tasks, namely to fetch a transaction from the queue, to wait for a *service time* amount of simulation time, and to output the transaction. Transactions have no particular class and flow through a queuing system unmodified (ref. Fig 4a).

The performance model is refined by implementing processing functionality in servers, i. e. adding transaction modification, creation, and deletion capabilities. In addition, different classes of transactions can be introduced that are distinguishable by the server.

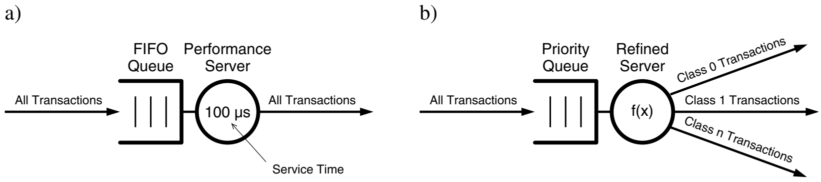


Fig. 4. Queuing system a) At performance level of abstraction (only data flow), b) At a lower level of abstraction (control and data flow)

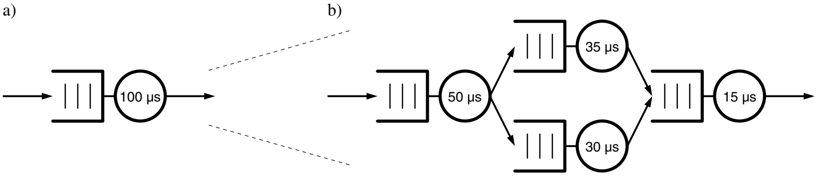


Fig. 5. Structural refinement by decomposition: a) Abstract system, b) Refined system

Figure 4b provides a refined example. In the figure, the refined server switches transactions to different output ports depending on their class. Functional refinement may even lead to systems that contain queuing systems which are represented by RTL models. In this case communication and/or structural refinements are likely to take place in conjunction with functional refinements.

4.2 Structural Refinement

Structural refinement decomposes queuing systems into queuing networks as illustrated in Figure 5. This form of refinement is often motivated by incorporating details of the architecture in the model.

Applying structural refinement we can address the granularity of existing building blocks using different abstraction levels in one model. This enables the simulation of mixed abstraction levels. In case of networking, for instance, an existing hardware CRC block can be instantiated within the performance model.

4.3 Communication Refinement

Communication refinement adds precision to communication between queuing systems or changes communication semantically.

On the performance level communication is non-blocking, instantaneous, and limited to reads and writes of classless transactions. This can be refined to blocking semantics. Timed or rated communication channels are another refinement option. Communication refinement is achieved using SystemC wrappers, comprehensively described in [5].

5 Case Study

To demonstrate our SystemQ approach we model a packet processing system for the network access. The system is able to aggregate traffic of up to 512 digital subscriber lines (DSL) and to forward this traffic to the upstream core network. In downstream direction the system distinguishes packets per DSL customer and forwards them to the appropriate output port. Figure 6 depicts the flow of packets through the processing system at block diagram level.

5.1 Setup

To compare levels of abstraction we define three different simulation models: 1) A pure performance model without any functionality and with only one class of transactions, 2)

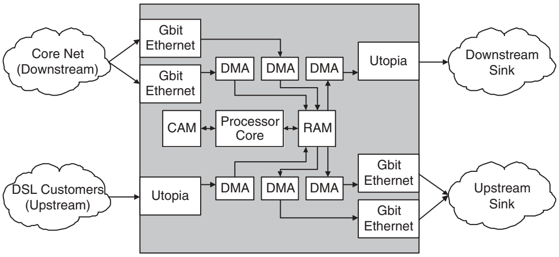


Fig. 6. Block diagram of a packet processing system

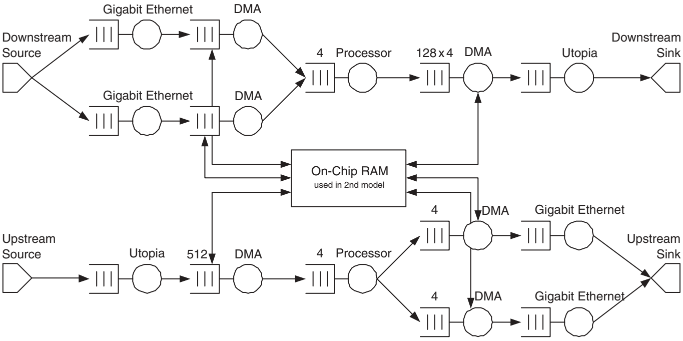


Fig. 7. SystemQ model of a packet processing system with structural refinement

a structurally refined performance model reflecting the influence of the on-chip RAM, 3) a functionally refined model that distinguishes four classes of transactions, i. e. ATM cells as well as short, medium, and long Ethernet frames. These transactions are processed differently.

Figure 7 shows the first two models. Every functional block in Fig. 6 is represented by a queuing system. Both models abstract from some parts of the processor, i. e. the CAM. As already shown in Figure 4a we do not distinguish classes of transactions for these two models. The models are completed by a testbench modeling traffic of up to 512 DSL customers for up- and downstream direction.

5.2 Results

Using the setup we evaluate our approach in terms of simulation performance, modeling effort, and quality of results. We use a standard 2.4 GHz Linux-based Intel system for the SystemQ simulation. As a reference a Mentor Graphics VStationPRO hardware emulation system is used running a register-transfer level model of the packet processing system.

Simulation Performance. Figure 8 shows the simulation performance of our different test cases. The performance difference between the performance model and the structurally refined model is not significant (about 6 %). However, performance model and

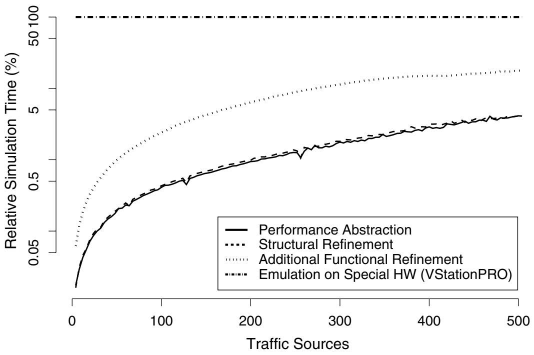


Fig. 8. Simulation performance of SystemQ models compared to emulation hardware

functionally refined model differ by a factor of about six. All performance results in the figure are scaled to the RTL hardware emulator. On average, our SystemQ performance model running on the standard PC is about 63 times faster than the hardware emulation. For the refined example the speedup is still 11 x.

Modeling Effort. An RT level model for the packet processing system can be built in a time frame of roughly 100 man-weeks using pre-built IP modules where applicable. Contrary to that the performance model was built in just one man-week from scratch using the SystemQ framework. The refinement of this model took about half a week for the structurally refined model and about two weeks for the functionally refined model, as shown in Figure 9.

Quality of Results. So far, we have compared results of the different SystemQ models (see Fig. 10). The figure exemplarily shows the utilization of a direct memory access (DMA) engine over the number of traffic sources for different abstraction levels. The initial performance model (solid line) exhibits a linear dependency for the considered interval since memory is unlimited. The structural model introduces memory limits. Utilization saturates at about 75 % because packets are dropped when the limit is reached instead of being fully processed by the DMA. The functionally refined model uses the same memory limits but requires a higher utilization of about 90 % due to the now explicit handling of packet descriptors. In this case, the functional refinement led to more accurate performance results, that in turn could be put back into the performance model.

5.3 Discussion

By defined refinement paths SystemQ provides a systematic approach to fast high-level modeling that seamlessly integrates with existing SystemC based methodologies, e. g., mixed level modeling. Already our preliminary results indicate reasonable support for system architects during early phases of the design flow:

Modeling effort. Introducing new abstractions layers, we have shown that turnaround times for early system evaluation can significantly be reduced by using Sys-

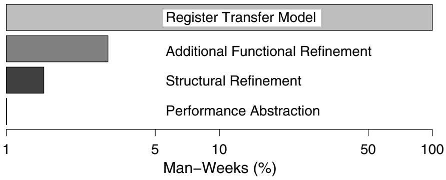


Fig. 9. Modeling effort of SystemQ models compared to a bit-true model (RTL)

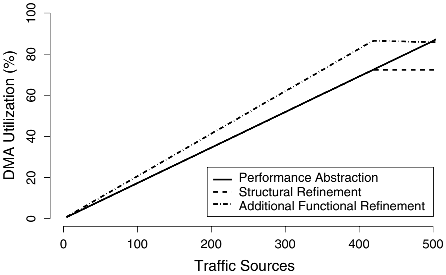


Fig. 10. Downstream DMA utilization monitored at different abstraction levels

temQ. In combination with a rich design library, this feature enables timely modeling of system-level design alternatives.

Simulation performance. By further reducing the degree of details SystemQ can achieve faster simulation performance than established transaction- and RT level models. The observed speedup factor of 63 x for our performance model compared to RTL HW emulation translates into several orders of magnitude for pure RTL simulation. In combination with low modeling effort, this aspect enables fast turn-around times for evaluation of system architecture instances.

Quality of results. Comparing the different abstractions for our models, we were already able to reveal the impact of structural and functional refinements on the quality of performance estimation. A more quantitative picture can be drawn after completing the refinement process down to RT level.

We currently determine the influence of different scheduling algorithms on our packet processing system. We expect SystemQ's formalism that exposes transaction scheduling and time- and workload-dependent behavior to be beneficial for the ongoing analysis.

6 Related Work

We find related work of system-level platform design in three areas of research.

Metropolis [6] is a very general framework where arbitrary models of computation can be expressed by using the meta modeling language. Architecture, function, and mapping aspects are represented by separated entities. Its generality however makes the modeling effort more complex and less intuitive than by restricting the designer to one consistent representation for particular application domains.

Artemis [7] and YAPI [2] are targeted at media processing (streamed) applications and are based on Kahn process network (KPN) representations. Functional and architecture models are handled separately and associated with each other using an explicit mapping step. For evaluation, execution events are communicated from the functional layer to the architecture layer. There is however no feedback path from architecture timing to the functional layer. Scheduling cannot be specified by a KPN, which is why extensions like virtual resources are required to enable scheduling analysis.

In the network domain, we recognize Click [8] as a modeling formalism for packet processing applications. Packet and resource scheduling are not part of the formalism and only one abstraction level exists. StepNP [9] is written in SystemC and uses Click as input for applications, whereas architecture building blocks are SystemC modules at RTL or transaction-level. Apart from StepNP, the work in [10] and the references therein describe the state-of-the-art of platform modeling in SystemC. Components can be represented at transaction and RT levels, and programmable components can be abstracted up to instruction set simulators. The simulation of several heterogeneous architecture building blocks at these levels is too complex to be used during concept phase.

7 Conclusion

We have presented a fresh approach to architecture performance modeling aimed at the concept phase of platform development. Our framework SystemQ is inspired by modern application domains not addressed by existing frameworks, such as network processing. SystemQ is based on the following principles:

- The specification of transaction scheduling disciplines is part of the chosen queuing systems modeling formalism.
- Timing information from the architecture model can be fed back into the functional model to consider time-dependent behavior of the application, as imposed by, e. g., network protocols and dynamic load balancing.
- SystemQ provides a path to implementation through refinement down to SystemC's established abstraction levels (transaction/RTL). SystemQ is realized by SystemC modules. It allows modeling on different abstraction levels by employing defined component interfaces.

Our ongoing case study of a network system has already shown that SystemQ is efficient enough to support designers during concept phase while providing a systematic refinement path by gradually adding functionality and structure.

Acknowledgments

The authors wish to thank R. Thudt and A. Schumacher for many helpful discussions. We also thank Prof. K. Franke (TU Chemnitz) for his invaluable support.

References

1. Keutzer, K., Malik, S., Newton, A.R., et al.: System level design: Orthogonalization of concerns and platform-based design. IEEE Transactions on CAD **19** (2000)
2. de Kock, E.A., Smits, W.J.M., van der Wolf, P., et al.: YAPI: Application modeling for signal processing systems. In: DAC. (2000)
3. Kleinrock, L.: Queueing Systems, Volume I: Theory. John Wiley & Sons (1975)
4. Baskett, F., Chandy, K.M., Muntz, R.R., Palacios, F.G.: Open, closed, and mixed networks of queues with different classes of customers. Journal of the ACM **22** (1975)

5. Grötker, T., Liao, S., Martin, G., Swan, S.: System Design with SystemC. Kluwer (2002)
6. Balarin, F., Watanabe, Y., Hsieh, H., et al.: Metropolis: An integrated electronic system design environment. IEEE Computer **36** (2003)
7. Pimentel, A., Hertzberger, L., Lieverse, P., et al.: Exploring embedded-systems architectures with Artemis. IEEE Computer **34** (2001)
8. Kohler, E., Morris, R., Chen, B., et al.: The Click modular router. ACM Transactions on Computer Systems **18** (2000)
9. Paulin, P., Pilkington, C., Bensoudane, E.: StepNP: A system-level exploration platform for network processors. IEEE Design & Test of Computers **19** (2002)
10. Wieferink, A., Kogel, T., Leupers, R., et al.: A system level processor/communication co-exploration methodology for multi-processor SoC platforms. In: DATE. (2004)

Moving Up to the Modeling Level for the Transformation of Data Structures in Embedded Multimedia Applications

Marijn Temmerman[1,2], Edgar G. Daylight[2], Francky Catthoor[2], Serge Demeyer[3], and Tom Dhaene[3]

[1] Karel de Grote-Hogeschool, Salesianenlaan 30, B-2660 Antwerp, Belgium
[2] IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
[3] Universiteit Antwerpen, Middelheimlaan 1, B-2020 Antwerpen, Belgium

Abstract. Traditional design- and optimization techniques for embedded devices apply local transformations of source-code to maximize the performance and minimize the power consumption. Unfortunately, such transformations cannot adequately deal with the highly dynamic nature of today's multimedia applications as they do not exploit application specific knowledge. We decided to go one step back in the design process. Starting from the original UML (Unified Modeling Language) model of the source code, we transform the UML model first before refining it into executable code. In this paper we present (i) the transformation of various UML models, (ii) a fast technique for the estimation of the high-level cost parameters that steer our transformations, and (iii) experiments based on three case-studies (a Snake game, a Tetris game and a 3D rendering engine) that show that our transformations can result in factors improvement in memory footprint and/or execution time with respect to the original model.

1 Introduction

In the past decade, we observe a shift from image-based to object-based design in the multimedia application domain. Hence, the efficient implementation of object-based – and thus dynamic and non-manifest – applications on embedded platforms is investigated by many researchers from the embedded-systems-engineering community.

Traditionally, embedded systems engineers start the mapping of an application from executable code by exploiting platform-related information extensively [1]. Unfortunately, the original high-level design at the conceptual modeling level was usually not developed taking into account hardware-related criteria. This approach results in suboptimal implementations on the embedded platform, because at the source-code level not all the inefficiencies at the modeling level can be removed any more. *In order to achieve larger reductions in system costs, we apply novel conceptual model level transformations that are steered by platform-related cost parameters.* As far as we know, this is the first work addressing this issue.

Several abstract modeling languages exist, but we selected the UML to illustrate our transformations at the conceptual modeling level, because UML is widely accepted as an industrial standard.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 445–454, 2005.

The performance[1] and energy[2] consumption of multimedia applications are dominated by the number of data transfers over the memory hierarchy of the platform [1,2]. Moreover, these cost factors depend on the size of the memory units. Therefore, the purpose of our UML transformations is to construct new conceptual models for the data-dominant part of the application. These UML models can then be refined into *concrete data structures* (CDS), expressed in executable C code, which are economically accessed and consume a minimum amount of memory space.

We realise our goal by exploiting application-specific knowledge, which is easier, and thus faster performed at the modeling level. Figure 1 shows three equivalent models for the well-known game Snake, which is available on many handheld devices. In the original model (a), the snake is modeled as a sequence of cells on the board. By exploiting the spatial and temporal locality of the snake cells when the snake moves on the board, we can compress adjacent cells into a snake segment without loss of information. This results in the second model (b), in which the snake contains fewer, but larger – in number of attributes – parts. By combining the segments into one object, namely a broken line defined by just its corner points (c), we systematically derive yet another model for the game Snake. We will apply similar UML transformations to our other two case studies.

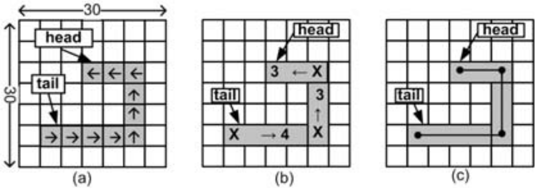


Fig. 1. Three models of the Snake game: (a) a snake modeled with cells, (b) a snake composed of segments, and (c) a snake modeled as a broken line

The multiple refinement possibilities, for each UML model, that exist at the CDS-level, create a large design space [3]. *Therefore, we propose an estimation technique for assessing the costs of a given UML design by counting the number of data accesses and calculating the memory footprint for a representative scenario and two particular concrete data structures.* Also that is a clear contribution of this paper.

In all our three case studies, we prove that our high-level estimates serve as reliable predictions for the actual implementations with respect to the memory footprint of the objects of the application, the number of data accesses (profiled with Atomium [4]), and the runtime (measured on the Trimedia) of the game engine for a complete, realistic game.

[1] Each data transfer over the memory hierarchy introduces a delay due to the memory latency.

[2] The memory-related energy consumption of a data structure can be estimated as A x J, where A is the overall data access count and J the energy of one data access. J depends on the memory size and technology.

Our approach has currently two drawbacks. First, the UML model, obtained after having applied UML-level transformations, and its implementation in executable code require an extensive effort from the designer. We will, in future work, focus on the formalization of the UML-transformations, allowing us to show how the transformations can be automated. Second, the techniques used in our UML-transformations are not generally applicable. Therefore, we focus on a specific application domain, namely 2D- en 3D object manipulation and rendering.

The rest of this paper is structured as follows: in Section 2 we start with related work. In Section 3 we discuss our UML-model transformations and explain our estimation technique on the basis of the Snake game. Section 4 presents two additional case studies that confirm the applicability of our UML-transformations for our application domain. We conclude in Section 5.

2 Context and Related Work

Traditionally, the data of multimedia applications is processed in a sequential pipeline consisting of three stages: the application engine, the frame buffer and the display. On a typical PDA platform the display subsystem (i.e. the frame buffer and the color LCD screen) consumes more than 50% of the total system power [5]. Hence, many researchers focus on the optimization of the display subsystem.

The power consumption of LCD displays has been tackled extensively at the hardware level [6]. Also, complementary to our UML-transformations, are the software-only optimization techniques for the LCD display power consumption of [7], in which the total system power is reduced by 50% on average.

Due to the fact that organic display systems illuminate themselves [8], both the authors of [5] and ourselves expect the power requirements of the frame buffer and associated busses to become more dominant. In addition to the hardware optimization techniques of [5], we can even omit the redundant frame buffer. We adhere to a nontraditional but increasingly popular, object-based rendering pipeline in which we model the whole application in terms of objects. These objects contain the information of both their functionality and rendering. In contrast to [5], the compression of the pixels is not performed at runtime, but at design time by our UML transformations.

In the embedded-systems community, research related to UML focus often on the interfacing of communicating objects [9] and do not consider data-related issues.

Our work is complementary and closely related to [3], which also aims at optimizing and exploring the data structures in multimedia applications. Nevertheless, there are two main differences. First, we concentrate on the optimization of data structures from a higher level of abstraction in the design flow. We go back to the UML modeling level for our transformations. The implementation into various concrete data structures is the subject of [3]. Second, our UML-model transformations focus on the exploration of the data-fields (i.e. the attributes) of the models of the objects and we do not perform optimizations at the CDS-level. The automation, and thus also the combination of our UML-model transformations with these at the concrete data structure level is subject to future work.

3 Exploration of the Design Space at the Modeling Level

Exploration of the design space – in memory footprint and number of data accesses – at the modeling level, consists in the systematic search for equivalent designs in which the models of the concepts differ (i) in the set of their primitive attributes and/or (ii) in the quantity of their composing parts. We reach our goals by exploiting geometrical and/or topological characteristics of the objects in the multimedia application.

In this Section we present our UML model transformations and propose a methodology for a fast comparison of the various models in terms of memory footprint and data accesses.

3.1 Transformations at the UML-Level

We selected the game Snake to demonstrate our UML model transformations. In this game a snake moves, under control of the player, on a board with a predefined speed. On the screen, the playing field is shown as a 2D grid of 30 by 30 positions. The snake is made up of equally colored cells and grows (i.e. cells are added) by eating food. The game ends when the snake's head collides with its body or with the border of the board.

We present three different UML-models of the game: (i) a cell-based model, (ii) a segment-based model, and (iii) a model based on the corner points of the shape of the snake. These UML models are conform to the three pictures of the Snake board from Figure 1.

Model 1: The Cell Model of the Snake Game Engine: The original UML model of the game engine (Fig. 2a) reflects the concepts as introduced in the description of the rules of the game and was extracted from existing source code. This game has one dynamic object: the moving and growing snake. All the information is stored in the `SnakeBoard` concept and the snake that is modeled as a collection[3] of `Cells`. It is not necessary to add the free cells explicitly in the model when the dimensions of the board are known.

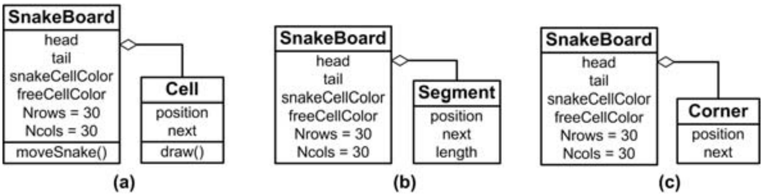


Fig. 2. (a) the UML-class diagram for the snake composed of snake cells, (b) the segment-based UML-class diagram, and (c) the UML-class diagram for the line-based model of the snake

The concept `Cell` has two attributes: (i) `position` stands for the location of the cell with its x- and y-coordinates on the board, and (ii) `next` is needed to store the

[3] A collection of objects is represented with a diamond shaped symbol in a UML class diagram [10].

dynamic structure of the snake and refers to the next cell in the snake. The attributes `head` and `tail` of the concept `SnakeBoard` indicate the first and the last cell of the snake. Exploitation of the fact that only the head and the tail cell need to be moved – because all the snake cells have the same color – results in an efficient algorithm for `moveSnake()` and the rendering process.

Model 2: The Segment Model of the Snake Game Engine: Careful observation of an average game shows that most of the time, the snake moves in the same direction. Hence, the shape of the snake contains only a few turns. We exploit this knowledge – i.e. the spatial and temporal locality of adjacent cells with the same moving direction – in the transformation of Model 1 to reduce the number of objects on the board without losing information. In this second model, we compress adjacent `Cells` into one `Segment` object (Fig. 2b). The `position` of a `Segment`, indicated with an x on the board, refers to the position of the first cell of the segment. The value of the attribute `length` equals the length of a `Segment` expressed in `Cell` units. Notice that the number of segments in the snake depends on its actual shape.

Model 3: The Line Model of the Snake Game Engine: In order to obtain Model 3 we combine the segments of Model 2 into one broken line (Fig. 2c). The snake is modeled with the sequence of the corner points of the line. A `Corner` object needs only its `position` on the board and a reference to the `next` corner point in the snake. The snake has one corner object more compared to the segment-based model, but we reduced the number of attributes by eliminating `length`.

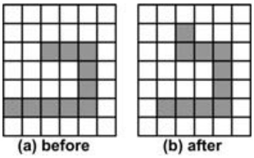
3.2 Evaluation of the UML-Models

The Proposed Estimation Technique. In order to compare the different designs at the modeling level in terms of the memory footprint and the number of data accesses[4], we need information from the lower abstraction level (i.e. after the refinement of the collections of objects).

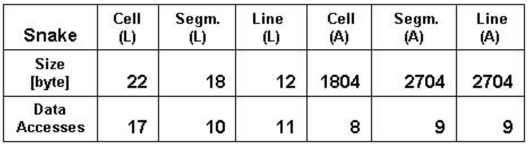
1. We consider two CDS-refinements, each situated at the extreme limits of the design space of the concrete data structures:
   (a) a List, which is very compact but requires sequential access
   (b) a large Array with direct access
2. We make abstraction from the dynamic and non-deterministic behaviour of the application by investigating a representative and dominant scenario.
3. By simulating this scenario for the two CDS-refinements of each UML model, we
   (a) calculate the memory footprint of the two concrete data structures
   (b) count the number of data accesses

   These numbers give us high-level estimates for the distribution of the implementations of the various models in the design space.

[4] To make abstraction of the memory hierarchy of the platform, we count the number of data accesses. A data access is an access of data whose size is between 1–8 Bytes [3].



(I) The scenario



Snake	Cell (L)	Segm. (L)	Line (L)	Cell (A)	Segm. (A)	Line (A)
Size [byte]	22	18	12	1804	2704	2704
Data Accesses	17	10	11	8	9	9

(II) High-level estimates for the Snake object

Fig. 3. (I) The scenario used for the evaluation of the UML-models of the Snake game: the snake consists of 10 cells and moves one position up on a board of 30 by 30 positions. (II) Comparison of the Snake models in terms of size and data accesses for this scenario with a List (L) and with an Array (A) for the concrete data structures

Evaluation of the Snake Models. We apply our methodology on the snake game.

1. For each collection of objects at the UML-level we consider two concrete data structures: (a) a List and (b) an Array. We use the attribute `position` as the key to identify an element (i.e. `Cell`, `Segment` or `Corner`) in the collection.
2. We investigate the scenario depicted in Fig. 3 I. The snake moves to its next position on the board and changes its moving direction.
3. The high-level estimates for this scenario are gathered in Fig. 3 II for both the List (L) and Array (A) refinements. This table also distinguishes between the three models: Cell, Segment, and Line model. For each model, we present the total actual size of the snake object (after the movement of Fig. 3 I). The calculation of the total size of the snake is based on the size of `tail` and `head`, and also the number of elements and their size (in bytes). The number of data accesses is obtained by counting all the read and write operations to the snake object conform to this scenario.

Extrapolating the high-level estimates for a complete game, we expect that the original `Cell` model design gives us a Pareto-optimal solution for the array implementation. We predict also two new Pareto points for the List versions of the `Segment` and the `Line` model.

3.3 Validation of the High-Level Estimates for Snake

We implemented six versions of the Snake game in the C language: CellL, CellA, SegL, SegA, LineL, and LineA. Figure 4 shows the profiling results of one complete, realistic game. We measured the execution time for the game engine on the Trimedia platform and obtained the data accesses from the analysis with the ATOMIUM tool [4].

With respect to the number of data accesses, the implementations CellA, SegL, and LineL are indeed Pareto-optimal solutions. The memory footprint of the Snake for LineL is a factor 53 smaller than for CellA. For this gain, LineL requires only 50% more data accesses than CellA. Trade-offs arise also between the memory size of the snake and the computing complexity of the algorithms. Our experiment shows that SegL and LineL, obtained by compressing the data, require an extra cost in execution time for the extraction of the detailed information. In this paper we focus on the data modeling aspects of the transformations leaving the computing complexity to future work.

Figure 5 shows that our high-level estimates serve as reliable predictions for the arrangement of the profiled implementations in the design space with respect to the

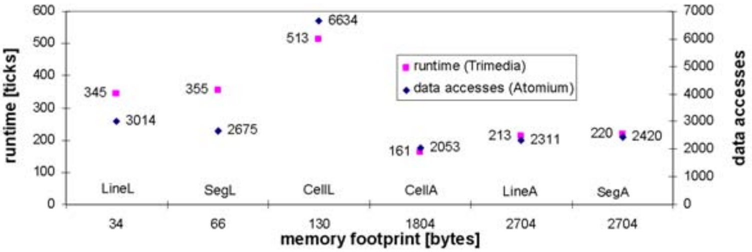


Fig. 4. Profiling results of a complete, realistic Snake game: runtime values of the game engine on the Trimedia and data accesses to the snake object from the Atomium analysis

memory footprint and the number of data accesses. The values in this table are calculated from the values from Figure 3 II and Figure 4. It is clear that the List (L) implementations of the segment-based and line-based models result indeed in two new Pareto-points (indicated in bold) in addition to the Array (A) refinement of the original cell-based model.

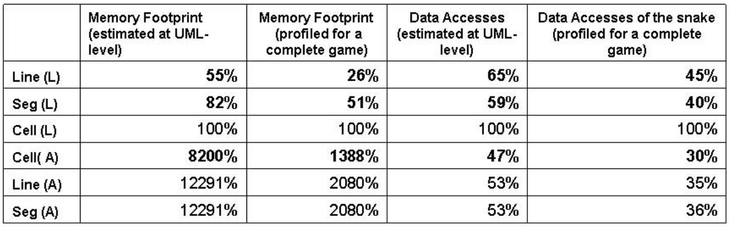
4 Additional Experiments

In this section, we present two case studies. We start with the 2D computer game Tetris. The second experiment concerns the model transformation for a 3D mesh.

4.1 Tetris Game

In Tetris, the player controls falling pieces, composed of four adjacent squared cells which appear in seven different shapes and colors. During game play, the landed pieces are piled up at the bottom of the board and form rows of cells in the pile (Fig. 6 I). Full rows are removed and the rows above move down. The game ends when the pile reaches the top of the board.

UML Model Transformations: From the initial Model 1 of the game engine, we construct two new models. Model 1 (Fig. 6 II) is the Cell-based model: both the `Piece`



	Memory Footprint (estimated at UML-level)	Memory Footprint (profiled for a complete game)	Data Accesses (estimated at UML-level)	Data Accesses of the snake (profiled for a complete game)
Line (L)	**55%**	**26%**	**65%**	**45%**
Seg (L)	**82%**	**51%**	**59%**	**40%**
Cell (L)	100%	100%	100%	100%
Cell(A)	**8200%**	**1388%**	**47%**	**30%**
Line (A)	12291%	2080%	53%	35%
Seg (A)	12291%	2080%	53%	36%

Fig. 5. Comparison of the estimates at the UML-level and the profiled values of the memory footprint of the snake object and the data accesses to the snake object for a realistic Snake game. Implementation Cell (L) is used as the reference

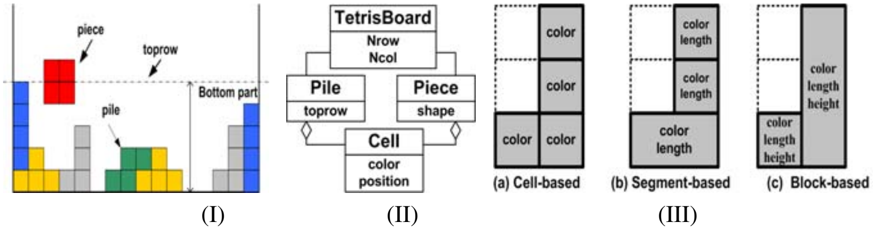


Fig. 6. (I) A frame of the Tetris game , (II) the UML-class diagram of the Cell-based Model, and a J-shaped piece modeled with (III a) 4 Cells, (III b) 3 Segments, and (IIIc) 2 Blocks

and the `Pile` object on the `TetrisBoard` are modeled as a collection of `Cells`. The attribute `shape` characterizes the `Piece` object. All the cells below `toprow` belong to the pile.

In the same way as for the snake game, we transform Model 1 to the segment-based Model 2 by compressing – in the falling pieces and thus also in the pile – the `Cells` on the same row into a `Segment` and to Model 3 by compressing the `Cells` into rectangular `Blocks` (Fig. 6 III).

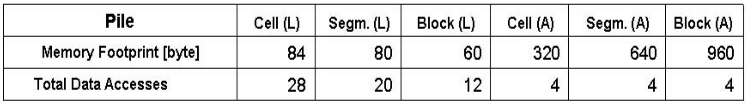
High-Level Estimates: Figure 7 shows the high-level estimates for the scenario depicted in Figure 6 I when the O-shaped piece moves one position down on the board. The board has 16 columns and 20 rows. We present for the `pile` object – the `piece` object is very small – the memory footprint and the number of data accesses, for the List and the Array refinements of the three UML models. The values in the table predict two Pareto-points: Cell (A) and Block (L).

Validation: The results from Figure 8 confirm our high-level estimates: the implementations CellA and BlockL give two Pareto-points, both in runtime (on Trimedia) and data accesses (Atomium analysis).

4.2 3D Mesh Rendering

A 3D mesh is traditionally constructed with triangular faces. Scenes with objects such as buildings or furniture, often contain many rectangular surfaces. We transform the model of the mesh by exploiting the co-planarity of the vertices at the modeling level.

UML Model Transformations: For this experiment we started from an existing software 3-D rendering API, designed for triangular meshes. By adding the class `Quad` to the



Pile	Cell (L)	Segm. (L)	Block (L)	Cell (A)	Segm. (A)	Block (A)
Memory Footprint [byte]	84	80	60	320	640	960
Total Data Accesses	28	20	12	4	4	4

Fig. 7. High-level estimates for the pile object with the scenario of Fig. 6 (I)

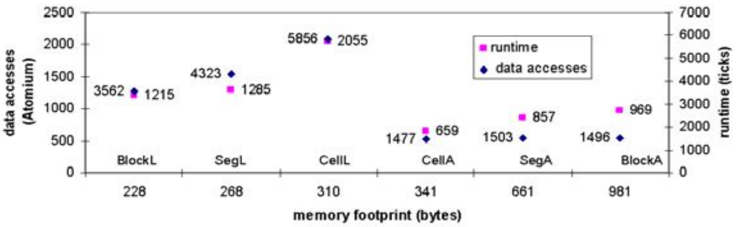


Fig. 8. Validation of the high-level estimates for a realistic game: runtime for the Tetris game engine on Trimedia and data access to the pile from Atomium

API, we are able to render meshes which also contain quadrangular faces. Figure 9 shows the UML-class diagram of (a) the triangle-based model and (b) the quadrangle-based model. A hybrid model could contain both types of faces. In this API, a mesh is implemented with two related Lists: (i) a List of the vertices with their 3-D coordinates, and (II) a List of the faces. A face object is implemented with the references (i.e. index) to its composing vertices from the first List.

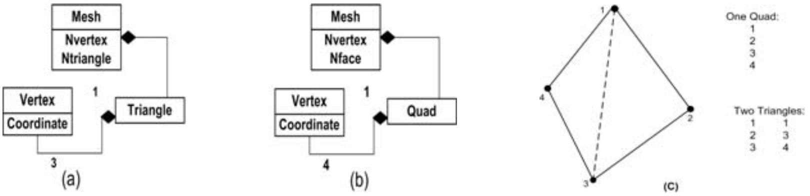


Fig. 9. (a) A mesh modeled with triangular faces and (b) a mesh modeled with quadrilateral faces, and (c) two triangles composing one quad

High Level Estimates: Both meshes have the same number of vertices, but the Quad Model needs half the number of faces compared to the original Triangle Model. The memory size of one face equals 16 bytes for a quadrangle and amounts to 24 bytes for the two composing triangles in the triangular-based mesh. This results in a gain of 34% for the memory footprint of the list of faces from the Quad Model. The rendering of the mesh requires a traverse of the list of faces and a lookup of the coordinates of each vertex of the face. To display one quad on the screen we need 5 data accesses to the mesh object, compared to 8 data accesses for the two composing triangles (i.e. a reduction of 38 % for the Quad Model).

Validation: For profiling we created first the data for a mesh composed of 3480 quadrangular faces and generated an equivalent mesh built up with triangles by splitting up each quad from the first mesh into two triangles. The application is implemented in C++ and executed on a Pentium IV platform under Windows XP. Again, the experiment confirms our high-level estimates. Compared to the mesh from the triangle-based model, the Quad mesh requires 78% for the memory footprint for the whole mesh object (i.e. vertices and faces). The profiled runtime for the rendering equals 481 msec for the triangle-based mesh and 349 msec for the quadrangle-based one (i.e. a gain of 28%).

5 Conclusions

We have presented UML model transformations for 2D and 3D multimedia applications steered by platform-related parameters. In this way we obtain new, non-trivial points in a Pareto space in which data accesses are traded off with memory footprint. The proposed high-level estimation technique allows a fast comparison of the various models without having to generate executable code. The UML transformations have been applied on three real-live case studies. Experiments on the Trimedia and with the Atomium tool validate our estimates at the UML-level.

References

1. Catthoor, F., et al.: Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design. Kluwer Academic Publishers, Boston (1998)
2. Vijaykrishnan, N., et al.: Evaluating integrated hardware-software optimizations using a unified energy estimation framework. IEEE Trans. Computers **52** (2003) 59–76
3. Daylight, E.G., et al.: Memory-access-aware data structure transformations for embedded software with dynamic data accesses. IEEE Trans. VLSI Syst. **12** (2004) 269–280
4. ATOMIUM: http://www.imec.be/design/atomium/ (2004)
5. Shim, H., et al.: A compressed frame buffer to reduce display power consumption in mobile systems. In: ASP-DAC. (2004) 818–823
6. Sarma, K.R., Akinwande, T.: Flat panel displays for portable systems. J. VLSI Signal Process. Syst. **13** (1996) 165–190
7. Gatti, F., et al.: Low power control techniques for tft-lcd displays. In: CASES. (2002) 218–224
8. Kamijoh, N., et al.: Energy trade-offs in the ibm wristwatch computer. In: ISWC '01: Proceedings of the 5th IEEE International Symposium on Wearable Computers. (2001) 133
9. Zhu, Q., et al.: System-on-chip validation using uml and cwl. In: CODES+ISSS. (2004) 92–97
10. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley (1998)

A Case for Visualization-Integrated System-Level Design Space Exploration

Andy D. Pimentel

Computer Systems Architecture Group,
Informatics Institute, University of Amsterdam,
Kruislaan 403, 1098 SJ, Amsterdam, The Netherlands
andy@science.uva.nl

Abstract. Design space exploration plays an essential role in the system-level design of embedded systems. It is imperative therefore to have efficient and effective exploration tools in the early stages of design, where the design space is largest. System-level simulation frameworks that aim for early design space exploration create large volumes of simulation data in exploring alternative architectural solutions. Interpreting and drawing conclusions from these copious simulation results can be extremely cumbersome. In other domains that also struggle with interpreting large volumes of data, such as scientific computing, *data visualization* is an invaluable tool. Such visualization is often domain specific and has not become widely used in evaluating the results of computer architecture simulations. Surprisingly little research has been undertaken in the *dynamic use* of visualization to guide architectural design space exploration. In this paper, we plead for the study and development of generic methods and techniques for run-time visualization of system-level computer architecture simulations. We further explain that these techniques must be scalable and interactive, allowing designers to better explore complex (embedded system) architectures.

1 Introduction

Chip technology continues to advance along the path predicted by Moore without any saturation in the exponential growth of transistor density foreseen within the next five years [1]. This growth is required to satisfy the demands of many diverse computer applications and has resulted in a steady increase in the complexity of today's computer architectures. A noticeable trend illustrating this increase in complexity is the move towards architectures that exploit parallelism at multiple levels of granularity (e.g. bit-level, instruction-level, and task-level) by partitioning the system into a multitude of specialized resources supporting a given level of parallelism. In the embedded systems domain, this trend is also clearly noticeable with the emergence of *Systems on a Chip* (SoCs) – or rather Multi-Processor SoCs (MPSoCs) – that can integrate an entire parallel system onto a single chip and start to play a vital role in the embedded systems market. The complexity of these MPSoCs is aggravated by the fact that – besides offering parallel computing resources – they often have a heterogeneous architecture, consisting of components that range from fully programmable processor cores to fully dedicated hardware blocks. Programmable processor technology is used for realizing

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 455–464, 2005.

flexibility, for example to support multiple applications and future extensions, while dedicated hardware is used to optimize designs in time-critical areas and for power and cost minimization. Because of the complexity of these MPSoC architectures, it is crucial to have good tools for exploring design decisions during the early stages of design. In recent years, much work has been undertaken in design space exploration and this paper explores an area that promises to increase the productivity of designers still further.

System-level simulation frameworks that aim for early design space exploration can create large volumes of simulation data in exploring alternative architectural solutions. Interpreting and drawing conclusions from these copious simulation results can be extremely cumbersome. In other domains that also struggle with interpreting large volumes of data, such as scientific computing, *data visualization* is an invaluable tool. Such visualization is often domain specific and has, surprisingly enough, not become widely used in evaluating the results of computer architecture simulations. Here, results are usually still presented graphically as a post-mortem but very little research has been undertaken in the *dynamic use* of visualization to guide design-space explorations. In this paper, we advocate the study and development of generic methods and techniques for run-time visualization of system-level computer architecture simulations. Specifically, we focus on those simulations that target architectural design space exploration. We explain that the visualization techniques must be scalable and interactive, allowing designers to better explore complex architectures that may be heterogeneous in nature and may exploit various levels of concurrency. To summarize, rather than presenting concrete research results, this paper tries to identify a challenging new research area.

The remainder of the paper is organized as follows. The next section provides an introductionary overview of the field of system-level design space exploration. In Section 3, we observe that hardly any research is performed on run-time visualization for architecture simulations, and that this is especially true from the perspective of design space exploration. In Section 4, we therefore plead for visualization-integrated design space exploration, in which generality, scalability, and interactiveness are key ingredients. Finally, Section 5 concludes the paper.

2 System-Level Design Space Exploration

The sheer complexity of modern (embedded) computer architectures forces designers to start with modeling and simulating system components and their interactions in the very early design stages. This is an important ingredient of *system-level design* [2, 3]. System-level models typically represent workload behavior, architecture characteristics, and the relation (e.g., mapping, hardware-software partitioning) between workload(s) and architecture. These models are deployed at a high level of abstraction, thereby minimizing the modeling effort and optimizing the simulation speed required to explore large parts of the design space. This high-level modeling allows for the early verification of a design and can provide estimates of the performance, power consumption and cost of a design.

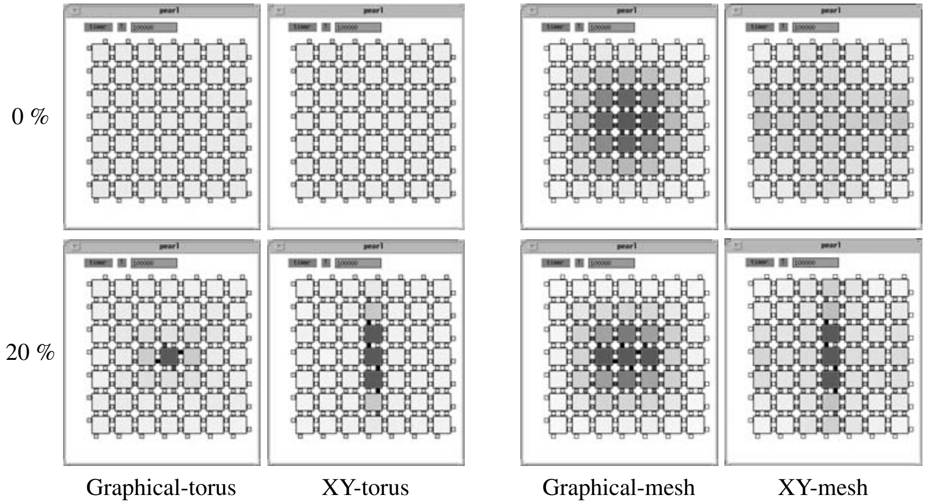
Design space exploration (*DSE*) plays a crucial role in system-level architecture design. It is imperative therefore, to have good evaluation tools for efficiently exploring different design choices during the early design stages, where the design space is largest. Consequently, considerable research effort has been spent in the last decade on developing frameworks for system-level modeling and simulation that aim for early architectural exploration. Examples are Metropolis [4], MESH [5], Milan [6], Artemis [7, 8] and various SystemC-based [9] environments such as the work of [10]. This research has produced significant results in various disciplines of system-level modeling and simulation. With respect to application modeling, or workload modeling, much work has been performed in the area of models of computation (e.g.,[11, 12, 13, 14]). System-level modeling and simulation of architectures and their performance constraints has been addressed by a large number of research groups (e.g., [4, 8, 15, 6, 16]). In many of these efforts, transaction-level models [17] are applied in which transactions between architecture components are modeled by atomic transfers of high-level data and/or control. Various research groups also recognize an explicit mapping step between application (workload) models and architecture models and subsequently proposed different mapping mechanisms (e.g., [8, 18, 4]).

Research on the refinement of (abstract) system-level architecture performance models to gradually disclose more implementation details is gaining interest but is still in its infancy. There are several attempts being made to address this issue, such as in the Metropolis [19], Artemis [8], and Milan frameworks [6], the work of [20], and in the context of SystemC (e.g., [10]). In [20], for example, a methodology is proposed in which architecture-independent specification models are transformed (i.e., refined) into architecture models to facilitate architectural exploration. The majority of the work in this field, however, focuses on communication refinement only (e.g., [21, 22, 23]).

Finally, different methods have been proposed for helping designers to quickly find good candidate architectures that can subsequently be further evaluated and explored by means of simulation. These methods usually apply multi-objective optimization techniques (e.g., [24, 25, 26, 27]), or in the case of [6], symbolic analysis.

3 Visualization, or the Lack of It

System-level simulations may exhibit vast amounts of simulation data on various characteristics (validity, performance, power consumption, reliability, etc.) for the architecture(s) under investigation. As mentioned before, interpreting and drawing the right conclusions from such copious simulation results may be extremely cumbersome. Because of exactly this reason, other domains that also struggle with interpreting massive amounts of data (or code), such as scientific computing, have embraced data (code) visualization (both at run-time and post-mortem) as a real aid for analysis and interpretation. As a result, visualization has become a research field in its own right in these domains. The same is not yet generally true for the computer architecture domain. Run-time visualization can, however, be extremely helpful to a system designer in identifying or analyzing dynamic effects that may occur during simulation and which may affect static performance but which cannot be analyzed at post-mortem. Here, one can think of, for example, synchronizations, cache behavior and coherency traffic in MPSoCs, or network contention and congestion in Network-on-Chip [28] based MPSoCs. To briefly illustrate the usefulness of run-time visualizations in the context of computer architecture simulation, Figure 1 shows a case study from some early visualization work by our group [29]. The pictures show visualizations of network simulations for eight different network configurations, i.e., for two types of networks (7×7 torus and mesh networks), two types of routing mechanisms (XY routing based on dimension ordering and Graphical routing based on Bresenham's line-drawing algorithm), and two network loads (a uniform network load (= 0% hotspot) and a network load in which 20% of the traffic is directed towards a hotspot in the middle of the network). The darker cells in Figure 1 indicate a higher network contention. The visualizations show at a glance the different behaviors of, for example, the two routing mechanisms. While Graphical routing performs well in a torus, it clearly suffers from problems in a mesh network.



Graphical-torus XY-torus Graphical-mesh XY-mesh

Fig. 1. XY and Graphical routing in torus and mesh networks

Despite the clear benefits, very little research is being conducted into generic methods and techniques for run-time visualization of (system-level) computer architecture simulations. Even more so, research on run-time visualization support to aid the DSE process is basically non-existing. Existing visualization work in the context of computer architecture simulation mainly focuses on visualization technology for educational purposes (e.g., [30, 31]), tightly couples visualization to one particular, often lower than system-level, architecture simulation environment (e.g., [32, 33, 34]), or only provides support for post-mortem visualization of simulation results (e.g., [35, 36]). To the best of our knowledge, only the recent research efforts of [37, 38] and especially [39] target generic visualization support in the domain of computer systems' analysis. Although the work of [37, 38] provides generic visualization support, it does so for a wide range of computer system related information which may not necessarily be applicable to computer architecture simulation, with its own domain specific requirements. Here, the data is generated dynamically and the goals are normally to refine a design space with minimum computer resources and elapsed time.

The Vista work from [39] aims at generic support for visualization of computer architecture simulations, but it does not target system-level simulation of systems that may comprise a large number of architectural components. As will be pointed out in the next section, this will have impact on the scalability requirements of the visualization. In addition, none of the above research efforts addresses the needs for visualization from the perspective of DSE, in which different trade-offs regarding, for example, performance, power, area, etc., need to be studied. Finally, the aforementioned visualization efforts do not provide the level of interactivity that is desired for effective DSE. In this respect, we envision a visualization-integrated DSE process in which the designer is allowed to provide (interactive) feedback to the simulation environment in order to actively explore and investigate the simulation results, or even to steer the simulation (as will be discussed later on). We believe that such visualization-integrated DSE is critical for improving the effectiveness of (future) system-level DSE approaches, which will eventually lead to reductions of design times.

4 Visualization-Integrated DSE

We plead for the development of *generic* methods and techniques to provide *scalable* and *interactive* run-time visualization of system-level computer architecture simulations for DSE. This section will shed some light on what we exactly mean with each of these requirements – generality, scalability, and interactiveness – for the run-time visualization methods. More specifically, we propose to evaluate visualization methods for DSE according to three quantifiable criteria, illustrated as three separate dimensions in Figure 2. Two of these dimensions relate to scalability, while the third refers to interactiveness. The aforementioned requirement of generality can be seen as a fourth criterion, but this one is less easy to quantify.

4.1 Generic Visualization Support

To optimize re-use of visualization building blocks, it must be identified what types of generic visualization building blocks are required to compose run-time visualizations for a wide range of computer architecture simulations. Because characteristics other

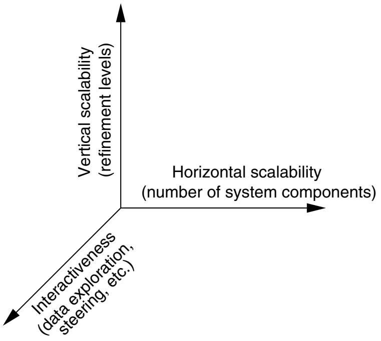


Fig. 2. Evaluation criteria for visualization methods for DSE

than just performance, like power consumption and even cost, also are major design goals in the embedded computing domain, it should be taken into account that a visualization can be applied to different dimensions of the data produced. That is, visualization building blocks must allow for effective employment in the context of performance analysis, the analysis of power consumption, etc. The choice of visualization building blocks is also influenced by the type of data values to be visualized. In this respect, a simple classification of two types of values can be made [29]: *snapshot* values that relate to a particular moment in simulation time (e.g., the signaling of a cache hit/miss) and *integrated* values that relate to the simulation over some time interval (e.g., the cache hitrate). In many occasions, a designer will be interested in, for example, performance behavior over some period of time. This may therefore require that snapshot values from the simulator are first transformed into integrated values before visualizing them. This can be established by defining a number a basic transformations on (raw) data values. Examples of such transformation are *smooth* (smooth a value by calculating the weighted average of the old smoothed value and the current value), *history* (keep a history of value samples), and *average* (calculate the average of a history of value samples). Flexible support for composing a wide range of such transformations on data before they are visualized, is therefore needed.

Furthermore, an efficient and effective mechanism, including a generic API definition, is needed that allows the placing of *probes* in an architecture simulator to capture the events or variables that need to be visualized. The probing mechanism should minimize intrusion and pollution of the target architecture simulation code. This could be achieved by devising a descriptive language with which a designer can describe how the events/variables captured from a simulator need to be *transformed* (applying the aforementioned transformations) and *visualized* (the type of visual used). We believe that an XML-based language may be a good candidate for such visualization descriptions since XML is already used extensively to describe the structure of (system-level) models [40, 41]. By applying visualization descriptions, we establish a strict *decoupling* of the visualization and the simulator code, which minimizes intrusion and pollution of simulator code and maximizes the potentials for re-use of the composed visualizations. The latter includes both re-use within a single simulation environment as well as sharing visualization components between different simulation environments. This should considerably improve the current situation in which designers typically need to develop their own proprietary visualization support.

4.2 Scalable Visualization

In the envisioned run-time visualization technology, the visualizations should be highly scalable, both *horizontally* and *vertically* (see also Figure 2). By horizontal scalability we mean that the visualization methods and techniques should be capable of visualizing simulations of architectures with possibly very large numbers of computational elements, memory and communication components (assuming that sufficient computational resources are provided to perform the visualization). This is an important criterion since future MPSoCs may scale up to systems that integrate hundreds to thousands of processing elements.

With vertical scalability, we refer to the capability of visualizing computer architecture simulations at multiple levels of abstraction. This is analogous to the gradual refinement of system-level architecture simulation models to exhibit more implementation details. It should, for example, be possible for visualizations to follow a similar refinement trajectory, i.e., gradually showing more detailed information. Therefore, it needs to be investigated whether or not such refinements can be formalized in transformations that move the visualization perspective through different levels of abstraction.

4.3 Interactive Visualization

Since the objective is support for DSE, the envisioned visualization technology should not be restricted to a one-way flow of information, namely from simulation to visualization. Rather, the designer should be able to provide interactive feedback to the visualization environment, thereby allowing the designer to actively explore and investigate the simulation results and maybe even *steer* the simulation. In our view, three different types of interactive feedback can be provided by a designer. First, a designer can change the view of a visualization. This might be done by changing the way data is visualized but retaining the same abstraction level, or by changing the level of abstraction in a visualization (as discussed in the above).

The two remaining types of feedback both deal with steering the simulations. This steering is based on the idea of *computational steering* [42, 43] that is commonly applied in the field of scientific computing. In the second type of feedback, simulations can be steered – or orchestrated – by interactively starting up (and stopping) parallel instances of a simulation with different parameters, according to the findings of the designer. With the proper support for visualization, these parallel instances of a simulation and, in particular, the differences between them, aid the architect in the DSE process. This steering mechanism is illustrated in Figure 3(a).

The third type of feedback, which is illustrated in Figure 3(b), comprises the steering of a simulation by changing its parameters at run-time. For (relatively) long-running simulations, it may be too time-consuming to start up a new simulation with differ-

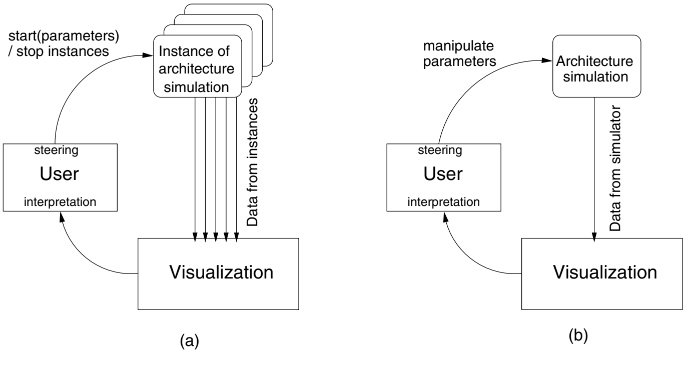


Fig. 3. Two types of steering in interactive visualization for DSE: (a) starting (and stopping) instances of an architectural simulation with different parameters, and (b) run-time manipulation of simulation parameters

ent parameters to reach a certain interesting point in execution again. Therefore, the potentials for manipulating a running architectural simulation (changing its parameters) also need to be investigated.

Finally, we would like to mention that both forms of interactive steering are orthogonal. So, they can complement each other in order to improve the process of DSE even further.

5 Conclusions

In this paper, we advocated the development of generic methods and techniques for run-time visualization of system-level computer architecture simulations. More specifically, we argued that especially visualization support to aid the process of architectural design space exploration deserves more attention. It was also explained that generality, scalability, and interactiveness are key ingredients in our envisioned visualization technology. Eventually, the proposed visualization-integrated design space exploration should lead to reductions in design times, and hopefully result in better designs. Of course, a logical next step is to give the ideas presented in this paper a more concrete form.

References

1. International Technology Roadmap for Semiconductors: Executive summary. http://public.itrs.net/Files/2003ITRS/Home2003.htm (2003)
2. Keutzer, K., Malik, S., Newton, A., Rabaey, J., Sangiovanni-Vincentelli, A.: System level design: Orthogonalization of concerns and platform-based design. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **19** (2000)
3. Gajski, D.D.: System Level Design Flow: What is needed and What is not. Technical report, CECS, University of California at Irvine (2002) CECS-TR-02-33.
4. F. Balarin et al.: Metropolis: An integrated electronic system design environment. IEEE Computer **36** (2003)
5. Cassidy, A., Paul, J., Thomas, D.: Layered, multi-threaded, high-level performance design. In: Proc. of the Design, Automation and Test in Europe (DATE). (2003)
6. Mohanty, S., Prasanna, V.K.: Rapid system-level performance evaluation and optimization for application mapping onto SoC architectures. In: Proc. of the IEEE International ASIC/SOC Conference. (2002)
7. Pimentel, A.D., Lieverse, P., van der Wolf, P., Hertzberger, L.O., Deprettere, E.F.: Exploring embedded-systems architectures with Artemis. IEEE Computer **34** (2001) 57–63
8. Pimentel, A.D.: The Artemis workbench for system-level performance evaluation of embedded systems. Int. Journal of Embedded Systems (2005)
9. Grötker, T., Liao, S., Martin, G., Swan, S.: System Design with SystemC. Kluwer Academic Publishers, Dordrecht, The Netherlands (2002)
10. T. Kogel et al.: Virtual architecture mapping: A SystemC based methodology for architectural exploration of system-on-chip designs. In: Proc. of the Int. workshop on Systems, Architectures, Modeling and Simulation (SAMOS). (2003)
11. Buck, J., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: A framework for simulating and prototyping heterogeneous systems. Int. Journal of Computer Simulation **4** (1994) 155–182

12. de Kock, E.A., Essink, G., Smits, W.J.M., van der Wolf, P., Brunel, J.Y., Kruijtzer, W.M., Lieverse, P., Vissers, K.A.: Yapi: Application modeling for signal processing systems. In: Proc. of the Design Automation Conference (DAC). (2000) 402–405
13. Stefanov, T., Kienhuis, B., Deprettere, E.F.: Algorithmic transformation techniques for efficient exploration of alternative application instances. In: Proc. of the 10th Int. Symposium on Hardware/Software Codesign (CODES'02). (2002) 7–12
14. Turjan, A., Kienhuis, B., Deprettere, E.F.: Translating affine nested loop programs to process networks. In: Proc. of the Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES). (2004)
15. Mihal, A., Kulkarni, C., Sauer, C., Vissers, K., Moskewicz, M., Tsai, M., Shah, N., Weber, S., Jin, Y., Keutzer, K., Malik, S.: Developing architectural platforms: A disciplined approach. IEEE Design and Test of Computers **19** (2002) 6–16
16. Lahiri, K., Raghunathan, A., Dey, S.: System-level performance analysis for designing on-chip communication architectures. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **20** (2001) 768–783
17. Cai, L., Gajski, D.: Transaction level modeling: An overview. In: Proc. of CODES-ISSS. (2003) 19–24
18. Živković, V., van der Wolf, P., Deprettere, E.F., de Kock, E.A.: Design space exploration of streaming multiprocessor architectures. In: Proc. of the IEEE Workshop on Signal Processing Systems (SiPS). (2002)
19. Densmore, D., Rekhi, S., Sangiovanni-Vincentelli, A.: Microarchitecture development via Metropolis successive platform refinement. In: Proc. of the Design, Automation and Test in Europe (DATE). (2004)
20. Peng, J., Abdi, S., Gajski, D.: Automatic model refinement for fast architecture exploration. In: Proc. of the Int. Conf. on VLSI Design. (2002) 332–337
21. Abdi, S., Shin, D., Gajski, D.: Automatic communication refinement for system level design. In: Proc. of the Design Automation Conference (DAC). (2003) 300–305
22. Lieverse, P., van der Wolf, P., Deprettere, E.F.: A trace transformation technique for communication refinement. In: Proc. of the 9th Int. Symposium on Hardware/Software Codesign (CODES). (2001) 134–139
23. Nicolescu, G., Yoo, S., Jerraya, A.A.: Mixed-level cosimulation for fine gradual refinement of communication in SoC design. In: Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE). (2001)
24. Haubelt, C., Mostaghim, S., Slomka, F., Teich, J., Tyagi, A.: Hierarchical synthesis of embedded systems using evolutionary algorithms. In: Evolutionary Algorithms for Embedded System Design. Kluwer Academic Publishers (2002)
25. Palesi, M., Givargis, T.: Multi-objective design space exploration using genetic algorithms. In: Proc. of the 10th Int. Symposium on Hardware/Software Codesign (CODES). (2002)
26. Thiele, L., Chakraborty, S., Gries, M., Künzli, S.: A framework for evaluating design tradeoffs in packet processing architectures. In: Proc. of the ACM/IEEE Design Automation Conference (DAC). (2002)
27. Erbas, C., Erbas, S.C., Pimentel, A.D.: A multiobjective optimization model for exploring multiprocessor mappings of process networks. In: Proc. of the IEEE/ACM CODES+ISSS Conference. (2003)
28. Benini, L., Micheli, G.D.: Networks on chips: A new SoC paradigm. IEEE Computer **35** (2002) 70–80
29. Kok, H.C., Pimentel, A.D., Hertzberger, L.O.: Runtime visualization of computer architecture simulations. In: Proc. of the Workshop on Performance Analysis and its Impact on Design (in conjunction with ISCA '97). (1997) 15–24

30. Marwedel, P., Sirocic, B.: Multimedia components for the visualization of dynamic behavior in computer architectures. In: Proc. of the Workshop of Computer Architecture Education (WCAE'03). (2003)
31. Yehezkel, C., Yurcik, W., Pearson, M., Armstrong, D.: Three simulator tools for teaching computer architecture: Easycpu, little man computer, and rtlsim. Journal on Educational Resources in Computing (JERIC) **1** (2001)
32. P. S. Coe et al.: A hierarchical computer architecture design and simulation environment. ACM TOMACS **8** (1998) 431–446
33. Berkbigler, K., Bush, B., Davis, K., Moss, N., Smith, S.: À la carte: A simulation framework for extreme-scale hardware architectures. In: Proc. of the IASTED International Conference on Modelling and Simulation. (2003)
34. Stolte, C., Bosch, R., Hanrahan, P., Rosenblum, M.: Visualizing application behavior on superscalar processors. In: Proc. of the Fifth IEEE Symposium on Information Visualization. (1999)
35. Fang, W., Wang, C.L., Zhu, W., Lau, F.: Pat: A postmortem object access pattern analysis and visualization tool. In: Proc of the Int. Workshop on Distributed Shared Memory on Clusters (at CCGrid 2004). (2004)
36. Hlavacs, H., Kvasnicka, D., Ueberhuber, C.W.: Clue — a tool for cluster evaluation. In: Distributed and Parallel Systems (DAPSYS). (2000) 61–64
37. Bosch, R., Stolte, C., Tang, D., Gerth, J., Rosenblum, M., Hanrahan, P.: Rivet: A flexible environment for computer systems visualization. Computer Graphics **34** (2000)
38. Bosch, R.P.: Using Visualization to Understand the Behavior of Computer Systems. PhD thesis, Stanford University (2001)
39. Mihalik, A.: Vista: A visualization tool for computer architects. Master's thesis, Massachusetts Institute of Technology (2004)
40. Lee, E.A., Neuendorffer, S.: MoML - a Modeling Markup Language in XML, version 0.4. Technical Report UCB/ERL M00/8, Electronics Research Lab, University of California, Berkeley (2000)
41. Coffland, J.E., Pimentel, A.D.: A software framework for efficient system-level performance evaluation of embedded systems. In: Proc. of the ACM Symposium on Applied Computing (SAC '03). (2003) 666–671
42. Mulder, J., van Wijk, J., van Liere, R.: A survey for computational steering environments. Future Generation Computer Systems **15** (1999)
43. van Liere, R., Mulder, J., van Wijk, J.: Computational steering. Future Generation Computer Systems **12** (1997)

Mixed Virtual/Real Prototypes for Incremental System Design – A Proof of Concept

Stefan Eilers and C. Müller-Schloer

Institute of Systems Engineering, System and Computer Architecture (SRA),
University Hannover, Germany
{eilers, cms}@sra.uni-hannover.de

Abstract. Design automation has continually moved towards higher system levels. In recent years it has become possible to model and simulate whole heterogeneous systems, containing hardware as well as complex software components, described on different abstraction levels, with a correct prediction of function and timing. The remaining problem, however, is to transform such a virtual prototype into the final real prototype. This transformation is usually not feasible in a single step. Intermediate versions consist of real as well as virtual subsystems. This paper explores the possibility of a step-wise transformation process (incremental system design) leading to the requirement to combine real subsystems with simulated ones (mixed virtual/real prototypes). The paper discusses the necessary real-time prerequisites in terms of simulation method, programming language, RTOS and the interface between real and virtual subsystems to realize this goal with today's computing platforms.

1 Motivation

Modern system designs use a distributed network of microcontrollers, each solving different complex tasks. These computing-nodes usually interact and solve tasks in a cooperative way. In practice, it is a difficult task to design distributed systems, even with just a view computing-nodes. It is almost impossible to debug many controllers simultaneously to track down problems, caused by intercommunication and timing problems. The Institute of Systems Engineering, System and Computer Architecture (SRA) has addressed this problem and developed a system simulation environment, called ClearSim-MultiDomain (ClearSim-MD), providing the advantage to develop such complex systems completely virtually. Target systems may contain microcontrollers including RTOS and application software, digital and analog components and interconnects like busses. ClearSim-MD simulates their functional and timing behavior. This system model is called ”virtual prototype”.

The goal of every development is a real system. Therefore it is not sufficient just to have a working virtual prototype. Due to unavoidable modeling inaccuracies, it may happen that a real prototype will not work as expected from the simulation. Due to the fact that testing in the real environment exhibits all disadvantages that we try to avoid with our simulation approach, solving this problem is a very time consuming task. Due to the fact, that current design methodologies do not address this problem, we propose

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 465–474, 2005.

the incremental design approach. The idea is to transform the virtual prototype to the real prototype in small steps [1, 2]. The system is cut into suitable subsystems, such that the communication costs are minimized. This transforms the originally purely virtual prototype into a mixed virtual/real prototype. This means that the simulation system has to simulate the remaining virtual components in real-time. If the mixed prototype shows any problems, it is now possible to isolate the faulty subsystem. By this so called incremental design approach it is possible to mix almost any combination of virtual and real components within a real-time simulation containing virtual as well as real components. This type of simulation is called Mixed Virtual/Real Simulation or Mixed Reality Simulation. Therefore, our goal is beyond the usual hardware- and software-in-the-loop techniques (see definition in [3]).

Mixed reality simulation is a challenging goal. To realize this concept for typical use cases of the automotive and automization industry, we expect simulation loop times between 500 us and 1 ms (for instance, a typical ABS system by WABCO expects a minimum time of 3 ms between measurement of the tire speeds and changing the pressure on the hydraulic break system [4] and dSpace characterizes typical loop-times for hardware-in-the-loop (HIL) of 1ms [5]).

The research work presented in this paper aims to show the feasibility of the idea and introduces an example implementation, based on standard PC hardware.

In this paper we will discuss, after a short review of the existing non-real-time system simulator ClearSim-MD (section 2) in section 3 a solution to make the C++ code real-time-safe. Next, we discuss in section 4 the RTOS solution used in this project (RTAI/LXRT), which allows for the coexistence of real-time and non-real-time processes.

Section 5 addresses the problems associated with the virtual-to-real interface. The remaining part of the paper presents results of the present implementation (section 6) and a conclusion and outlook (section 7).

2 ClearSim-MD: Completely Virtual

ClearSim-MD allows to simulate virtual prototypes with their functional and timing behavior. These virtual prototypes contain several components on different levels of abstraction, written with different modeling languages. The philosophy behind this is to use the best modeling language for each subsystem. For instance, an extended finite state machine (EFSM) may be best to describe a digital component, but Modelica [6] is more appropriate for analog components. ClearSim-MD is open for any modeling language. So far we have integrated C/C++, Java, EFSM, Modelica, VHDL, MATLAB/Simulink and SPS. Even high-level description languages are available, like UML-Statecharts or SDL [7].

The main goal of ClearSim-MD is to assist developers to write software for embedded systems. Thus, it provides models of microcontrollers (like Infineon C-167, C-505, ARM), which execute the real software as binaries with accurate functional and timing behavior. With these models, the developer is able to predict whether his software has the correct functional and timing behavior in the current context of environment [8].

To create a virtual prototype, one has to connect all these subsystems or simulation modules. This is realized by a universal portable simulation interface, called UPSI [9], and a simulation kernel, which handles communication and timing, as discussed in the next section.

2.1 Simulation Kernel

The simulation kernel has to manage two things:

1. Synchronization of the timing between the simulation modules.
2. Transmission of events between simulation modules.

Event based simulation algorithms, whether optimistic or conservative, are widely used in many simulation systems. They provide very accurate results at maximum speed. Especially for feed-back systems and optimistic simulation algorithms, simulation-modules have to own the ability to rollback its internal state to a previous one. If this isn't possible, incorrectness is unavoidable due to the fact that simulation modules are simulated sequentially and therefore their local clocks are not incremented at the same time [10].

This kind of optimistic event-based timing synchronization is possible only in completely virtual simulation environments and used by ClearSim-MD. Real-time simulation has to increase the time monotonically. Therefore, rollback and other mechanisms of event-based simulation systems cannot be used. This problem will be discussed in the next section.

3 ClearSim-MD: Mixed Reality

A simulation, containing real and virtual components, has to satisfy the following requirements:

1. The simulation speed itself has to be increased to support complex virtual prototypes within given time limits.
2. The virtual simulation time has to increase monotonically and synchronously with real-time. Jumping into the ”future” or ”past” is not possible, as event-based systems do.
3. An interface between real and virtual components has to be realized, afflicted with very low latencies.
4. All virtual components have to be simulated within fixed time limits, according to the definition of hard real-time (see DIN 44300).

Most of the expected speed improvement (1) was realized by using current computer systems with their extremely boosted performance and by optimizing the simulation modules and their communication overhead. Due to the fact that we have to consider communication overhead, the goal is to simulate the virtual part of the system with a real-time factor[1] below 1. The monotonically increase of the simulation time is handled

[1] $R = \frac{t_{sim}}{t_{real}}$;$t_{real}$: Simulated virtual time; t_{sim}: Time needed to simulate the time t_{real}.

by a timer based simulation algorithm, which is out of the papers focus. The interface between virtual and real subsystems (3) is discussed in section 5.

The last requirement (4) lead to a hard real-time implementation of ClearSim-MD which will be discussed in the next section.

3.1 Hard Real-Time Environment

ClearSim-MD had to be modified to satisfy the requirements, defined above.

The software is written in C++ and portable to various operating systems, using an abstraction layer. This abstraction layer was adapted to support the new C++ middleware for real-time operating systems, introduced in section 4. The goal was to migrate the existing C++ code to the new real-time constraints, with as few modifications as possible. The original simulation kernel and its simulation modules were written without real-time requirements, therefore we had to take care to detect non real-time compliant code and modify it to meet our requirements. These modifications were realized by redefining and replacing commonly used C++ classes and C functions with real-time compliant releases which are implemented by the C/C++ Middleware, as introduced in the next chapter.

4 C/C++ Middleware for Real-Time Operating Systems

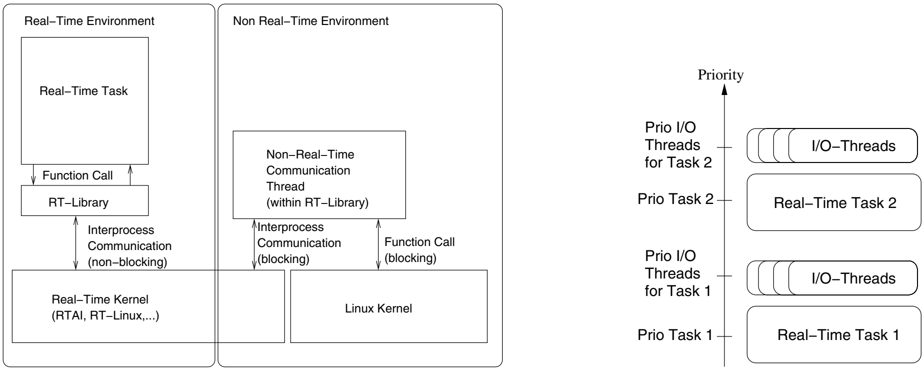
A general purpose operating system with real-time extension is used for ClearSim-MD with real-time simulation. Linux with RTAI/LXRT (http://www.aero.polimi.it/˜rtai/) provides the advantage of coexisting real-time and non real-time processes in user space.

All C real-time functions and C++ classes, as introduced in the previous sections, use an API, which is provided by this middleware. For mapping commonly used C functions easily to the real-time system, it was essential to implement this API close to commonly used C functions. Simple real-time extensions do not have features like file and socket access, therefore they had to be implemented by this middleware concept.

The solution was to run the simulation threads under real-time constraints, and let them communicate with a non real-time communication handler without blocking. This concept is based on threads which handle the data-communication between the real-time task and the non real-time system via non-blocking interprocess-communication, as for instance, pipelines or mailboxes.

In this implementation, a thread will receive data from the real-time interprocess communication and exports it into the corresponding output channel, like a file or socket handle. This thread runs in the non real-time environment, and may be blocked, either by missing data from the interprocess-communication or by the non real-time environment (see Fig. 1a). It is recommended that the communication thread runs with a slightly higher priority than the real-time task to prevent priority inversions. Thus, the communication is handled as fast as possible. Due to the fact that blocking of this communication thread does not block the real-time task[2], we can assure real-time constraints (see Fig. 1b) but have to accept loss of data if the internal buffers are full.

[2] If the communication thread is blocked by the non real-time system, the real-time task is reactivated by the real-time scheduler.


Real–Time Environment
Non Real–Time Environment
Real–Time Task
Function Call
RT–Library
Interprocess Communication (non–blocking)
Non–Real–Time Communication Thread (within RT–Library)
Interprocess Communication (blocking)
Function Call (blocking)
Real–Time Kernel (RTAI, RT–Linux,...)
Linux Kernel
Priority
Prio I/O Threads for Task 2
I/O–Threads
Prio Task 2
Real–Time Task 2
Prio I/O Threads for Task 1
I/O–Threads
Prio Task 1
Real–Time Task 1

(a) Structure of the rt-middleware, implemented as a rt-library

(b) Priorities of real-time tasks and I/O threads

Fig. 1. Structure and priority diagram of the real-time middleware architecture

Data input from non real-time into the real-time task needs a similar mechanism. Practical tests have shown that it is very useful to use a non real-time thread for every input channel which needs a slightly higher priority than the reading real-time task. This thread reads the data - for instance from a file - and transmits it, via non-blocking interprocess communication, to the real-time task.

Beside using the real-time functions for communication, we also need dynamic memory handling. To provide dynamic memory access, we need a special memory handler which never blocks and which provides a deterministic time behavior. It is quite clear that the memory management algorithms of general purpose operating systems are not feasible under real-time constraints [11].

For granting real-time constraints, there exist several optimistic and conservative algorithms for concurrent real-time memory management [12].

ClearSim-MD expects dynamic memory management. The introduced real-time middleware concept implements a simple conservative algorithm and optionally provides access to the implementation which may be integrated in the used host real-time environment. Thus, it is granted that dynamic memory handling is possible, even if the underlying real-time environment may not support it.

Using a real-time operating system is the base for providing real-time simulation. On top of this operating system, an adequate simulation is required, which will be shown in the next section.

5 Virtual/Real Interface

Providing a mixed virtual/real simulation environment, real components have to be connected to the simulation system.

Timing-correct simulation means that timing is part of the functional model of the simulation models. Therefore, it is essential, that timing is as correct as possible, avoiding incorrect functional behavior. As increasing abstraction always means to loose information, we have to take care that we will not loose the kind of information which is essential for the correct behavior of our target system.

Cutting a virtual prototype into subsystems, which are candidates for real or virtual components, may lead to different abstraction levels between the virtual and the real components. If the real and the virtual representation or abstraction level is the same, the connection is trivial. A usual interface on a low abstraction level has to convert physical values, like voltages, to their digital representation and vice versa. This means, virtual events are converted to their real representation as the event time is equal to the real-world time. And, for instance, changing voltages are converted to events with the current real-world time as event time. For this simple kind of connections, we provide a generic interface with very low latency, see section 6.2.

For instance, if a CAN-Bus is cut into a virtual and a real part for connecting real and virtual CAN-controllers to it, the bus will be cut on two abstraction levels: The virtual bus, which consists of a logical simulation of the bus communication with a timing model, and the real bus, which is just a wire, transporting voltage where timing exists just inherently. For timing-correct behavior, the virtual simulation environment needs information about the timing of the complete CAN bus, which now consists of a high level or virtual and a low level or real part. For extracting the timing information the complete bus has to be observable. Unfortunately, the timing information of the real part is „out of view" for the virtual system as this information is lost by converting the low-level bus communication to the higher level. There exist two solutions to handle this problem. The first one would be to lower the abstraction level of the virtual system, which will reduce the simulation speed and endanger a real-time simulation. The other solution is to create the information which was lost, accepting a higher degree of inaccuracy.

In this implementation, the interface for connecting a real CAN bus to the simulation system consists of a simple CAN controller. Due to the fact that the used abstraction level of the virtual model expects a complete message, the developer has to face the time needed to transport a complete message over the bus into the used CAN controller. This latency may change the order of messages on the bus, compared to the complete virtual simulation and is not preventable. In fact, as real controllers are not running in sync and are not always deterministic in its timing behavior, this may even happen on a real system. A wrong behavior caused by this „jitter" may indicate a wrong implementation and will happen on the complete real prototype, too. This problem needs to be discussed further, but exceeds the scope of this paper.

In the next section it will be shown that it is possible to use a virtual/real interface containing a timing model, which provides the lost information with an acceptable degree of accuracy for a correct simulation of the mixed prototype.

6 Validation of Concept and Implementation

To validate the implementation Code audits were essential but not sufficient. Using a synthetic simulation project, the correct implementation had to be validated practically as shown in the next section, followed by a section discussing the validation of the virtual/real interface.

6.1 Validation of the Real-Time Simulation Environment

To show the usability and correctness of the implementation, the ported simulation system ClearSim-MD was used to simulate a synthetic project with two communicating extended finite state machines (EFSM). This simulation was not designed to be comparable to any real world prototypes, but to find implementation errors. Two models which have to work symmetrically if they are running in a correct manner, would show implementation errors by asymmetrical anomalies in the timing analysis of the simulation environment. Each EFSM runs as single simulation subsystem within the simulation environment ClearSim-MD, controlled by the timer based simulation kernel with a cycle time of 1ms. Every EFSM receives data, does some calculation to generate CPU load (1000 divisions) and sends information to the other EFSM afterwards.

The simulation run was executed twice, the first time with the Linux system idle. During the second simulation run, the Linux system was under heavy load while hundreds of concurrent GCC-compilers were running on the system. Thus, the system was nearly unusable due to heavy processor and I/O load[3]. The Figures 2a and 2b show the plot of both executions. The y-axis is scaled to the needed cycle time of each simulation loop, the x-axis to the runtime of the simulation (5 seconds). The maximum allowed time of a simulation loop was set to 1 ms (cycle-time preset). Thus, no simulation cycle was allowed to violate this maximum delay, otherwise no hard real-time simulation was assured!

Figure 2a shows the timing behavior of the simulation while the system is idle. The little spikes of the I/O-time are caused by the communication between the simulation subsystems. Figure 2b shows the same situation under heavy load. The jitter of about 100 μs shows that there does exist an influence by the Linux system, caused by cache flushes and monolithic bus activity, but the maximum allowed cycle time is never reached or exceeded.

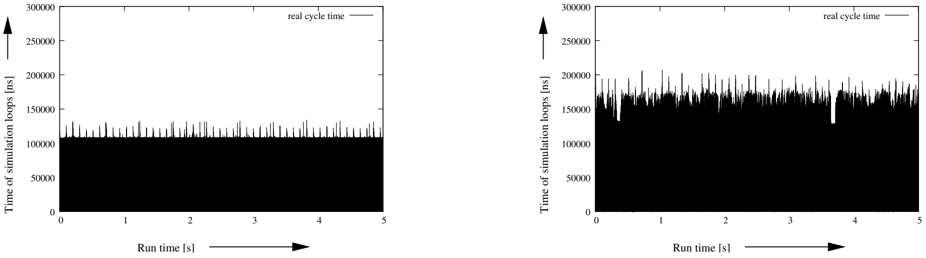
It could be shown that the real-time memory system stayed reliable in this extreme situation, while hundreds of running compilers were forced to swap a lot of memory to the harddisk.

With this simple example, the real-time capability of the ported simulation system could be evaluated successfully but does not show anything about the maximum simulation load possible. Although, this example is just showing a synthetic simulation model, it shows practically how the developer has to evaluate the timing behavior of his virtual prototype. In this example, he could safely reduce the loop time to about 300-250 μs without violating the assumed real-time behavior.

6.2 Virtual/Real Interface for Mixed Simulation

Minimizing delays by converting signals from virtual to real and vice versa, is essential for compareable results between the complete virtual and the mixed simulation.

[3] The response for user input by the non real-time system was nearly inexistent!



(a) Demonstration of a real-time simulation on an idle system. The cycle time preset is set to 1 ms

(b) Demonstration of a real-time simulation on a heavily loaded system. The cycle time is preset to 1 ms

Fig. 2. The time required for the time slice never comes close to the available cycle time of 1 ms, even when considering the spikes

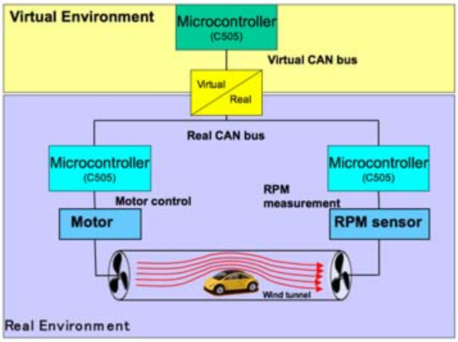


Fig. 3. Simulation of a mixed virtual/real prototype

Professional industry PCs with hardware input/output interfaces provide very low latencies and high computing power. Measuring the time needed to react to an external digital input and to create an external output, resulted in typical latencies of about 10 μs, which is much lower than expected. Additional delays are expected by conversion of the signals into simulation events, which will not be the dominant part as the speed of the processors is increasing continuously.

Simulation cycles of about 500 μs to 1 ms are the desired range of the systems under consideration. This will be feasible according to our results, even when using signals on high abstraction levels. Beside a high level interface as introduced above for the CAN-Bus, a generic interface for low level signals is provided for connecting electronic components, like motors, sensors and so forth. For using this interface, the developer has to connect the signal channels within the system description file and to configure it according his needs for instance to set the voltage range.

To show that it is possible to create a virtual/real interface on higher abstraction levels with correct timing behavior, a realistic simulation of a mixed prototype consisting

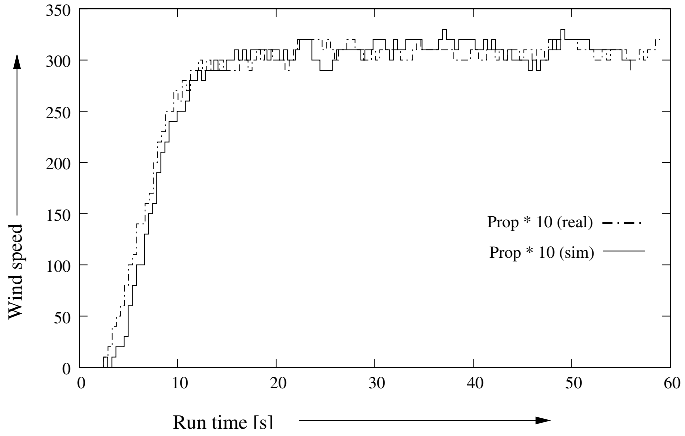


Fig. 4. Comparison of the simulated air speed as a function of time between completely virtual and mixed simulation. Both simulation agree within the expected error margins. In separate experiments, the overall agreement of the simulation results with the real wind tunnel experiment was also shown

of 3 microcontrollers (C505, a 8 Bit microcontroller by Infineon; one virtual, the other real) connected to a CAN bus, was used to control a wind tunnel, as shown in Fig. 3. This example was introduced in a previous paper [1] but just executed there without any real-time capabilities. Now, we are able to show that a functional and timing-correct simulation was possible within a real-time simulation of a mixed prototype (see Fig. 4). The virtual/real interface for the CAN bus contained a model to provide the essential timing information which was lost by converting messages from the real CAN bus to the virtual one, as discussed in section 5. While this example works well with one virtual C-505 microcontroller within real-time, the incremental design approach expects to export the wind tunnel first, then the microcontrollers for motor-control and RPM measurement.

Unfortunately, we currently have not enough power to to simulate 3 microcontrollers simultaneous within real-time constraints as we had to face peek times of 1,5 ms on our 2 GHz Pentium system while we expected a loop time of 1 ms. Therefore, further speed improvements need to be found to show the incremental approach on this platform successfully. For instance, timer based simulation allows for efficient parallelization of subsystem simulations which will provide further speedups. Another important next step is to find further real world examples to elaborate a relation between maximum model complexity and hard real-time capability.

7 Conclusion and Outlook

ClearSim-MD is a system simulator that allows to simulate complex systems, containing digital and analog components as well as microcontrollers and busses, using the optimal description language and abstraction level needed for a successful model of the system and its environment.

By transforming ClearSim-MD into a hard real-time simulation system, it was possible to close the gap between the completely virtual and the completely real prototype, simulating a mixed environment consisting of virtual and real components. We call this approach incremental design.

Although the developer has to take into consideration additional issues, like conversion latencies and lost timing information by cutting different abstraction levels, this approach enables developers to analyze and fix possible modeling and timing problems, which tend to be visible just on the real prototype and therefore may be hard to find in a virtual prototype. Especially on distributed and redundant computing architectures, an innovative, incremental simulation approach will be essential to handle the future complexity we will face.

With our example implementation, we were able to realize the idea of the incremental design approach. The computing power of the simulation host currently dictates the possible complexity of the virtual prototype to gain a simulation which stays within hard real-time constraints and stays synchronous with the real world clock. Influences, like cache latencies and concurrent bus accesses, which increase the jitter regarding to the system load, are not avoidable, but could be taken into consideration by analyzing the timing protocols of the simulation system. The next step is to test this concept on different real word examples, to analyze where the limits of model complexity currently are and how we could extend them. The future increase of processor power will help to speed up simulation. This proof of concept implementation will be the bases of further investigation, how a applicable incremental design methodology has to look like.

References

1. Eilers, S., Müller-Schloer, C.: Inkrementeller Entwurfsansatz mit Clearsim-RealTime. In: ASIM, Paderborn, SCS-Europe (2001) 205–210
2. Krisp, H., Bruns, J., Eilers, S., Müller-Schloer, C.: Multi-domain simulation for the incremental design of heterogeneous systems. In: ESM, Prag (2001)
3. Sailer, U., Essers, U.: Nutzfahrzeug-Echtzeitsimulation auf Parallelrechnern mit Hardware-in-the-Loop. Expert Verlag (1997)
4. Kruse, A.: Kopplung des physikalischen Simulators dSpace mit dem Systemsimulator ClearSim. Institute of Systems Engineering, System and Computer Architecture (SRA) (1997)
5. dSPACE `http://www.dspace.com`: Webside of dSPACE. (2005)
6. Tiller, M.: Introduction to Physical Modeling with Modelica. Springer (2001)
7. van Beek, D., Rooda, J.: Multi-domain modelling, simulation, and control (2000)
8. Eilers, S., Krisp, H., Müller-Schloer, C., Welge, R.: Inkrementeller entwurf verteilter, eingebetteter systeme mit vista. In: APC 2001. (2001)
9. Scherber, S., Müller-Schloer, C.: Entwicklungsumgebung zur modellierung und simulation heterogener mechatronischer systeme. In: Proc. Workshop Multi Nature Systems 99, Univ. Jena (1999)
10. Scherber, S.: Modellierung und Simulation software-intensiver eingebetteter Systeme. PhD thesis (2001)
11. Nilsen, K.D., Gao, H.: The real-time behavior of dynamic memory management in c++. In: IEEE Real-Time Technology and Applications Symposium, IEEE Computer Society (1995) 142–153
12. Ford, R.: Concurrent algorithms for real-time memory management. Software, IEEE **5** (1988) 10–23

Author Index