Lecture Notes in Computer Science 3793

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

Tom Conte Nacho Navarro
Wen-mei W. Hwu Mateo Valero
Theo Ungerer (Eds.)

High Performance Embedded Architectures and Compilers

First International Conference, HiPEAC 2005
Barcelona, Spain, November 17-18, 2005
Proceedings

Springer

Volume Editors

Tom Conte
NC State University, USA
E-mail: conte@ncsu.edu

Nacho Navarro
Universidad Politecnico de Catalunya, Barcelona, Spain
E-mail: nacho@ac.upc.edu

Wen-mei W. Hwu
University of Illinois at Urbana-Champaign, USA
E-mail: w-hwu@uiuc.edu

Mateo Valero
Universidad Politecnico de Catalunya, Barcelona, Spain
E-mail: mateo@ac.upc.edu

Theo Ungerer
University of Augsburg, Germany
E-mail: ungerer@informatik.uni-augsburg.de

Library of Congress Control Number: 2005936049

CR Subject Classification (1998): B.2, C.1, D.3.4, B.5, C.2, D.4

ISSN 0302-9743
ISBN-10 3-540-30317-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-30317-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11587514 06/3142 5 4 3 2 1 0

Preface

As Chairmen of HiPEAC 2005, we have the pleasure of welcoming you to the proceedings of the first international conference promoted by the HiPEAC Network of Excellence. During the last year, HiPEAC has been building its clusters of researchers in computer architecture and advanced compiler techniques for embedded and high-performance computers. Recently, the Summer School has been the seed for a fruitful collaboration of renowned international faculty and young researchers from 23 countries with fresh new ideas. Now, the conference promises to be among the premier forums for discussion and debate on these research topics.

The prestige of a symposium is mainly determined by the quality of its technical program. This first program lived up to our high expectations, thanks to the large number of strong submissions. The Program Committee received a total of 84 submissions; only 17 were selected for presentation as full-length papers and another one as an invited paper. Each paper was rigorously reviewed by three Program Committee members and at least one external referee. Many reviewers spent a great amount of effort to provide detailed feedback. In many cases, such feedback along with constructive shepherding resulted in dramatic improvement in the quality of accepted papers. The names of the Program Committee members and the referees are listed in the proceedings. The net result of this team effort is that the symposium proceedings include outstanding contributions by authors from nine countries in three continents.

In addition to paper presentations, this first HiPEAC conference featured two keynotes delivered by prominent researchers from industry and academia. We would like to especially acknowledge Markus Levy and Per Stenström for agreeing to deliver invited lectures.

The Levy lecture focused on the development of multicore processor benchmarks that address both heterogeneous and homogenous processor implementations. The Stenström lecture covered new opportunities and challenges for the chip-multiprocessing paradigm. They both provided us with insight into current technology and new directions for research and development in compilers and embedded systems.

Many other people have contributed greatly to the organization of HiPEAC 2005. The Steering Committee members provided timely answers to numerous questions regarding all aspects of the symposium preparation. Josep Llosa, Eduard Ayguad and Pilar Armas, the local Chairmen and Financial Chair, covered many time-consuming tasks of organizing a symposium: hotel negotiation, symposium registration and administration. We thank Sally McKee for the publicity, and Michiel Ronsse for the website and support for the PC meeting. Many thanks also to the Publication Chair Theo Ungerer, his scientific assistants Jan Petzold

and Faruk Bagci for volume preparation, and to Springer for publishing these proceedings as *Lecture Notes in Computer Science*.

We would like to also note the support from the Sixth Framework Programme of the European Union, represented by our Project Officer Mercè Griera i Fisa, for sponsoring the event and the student travel grants.

Finally, we would like to thank the contributors and participants, whose interest is the reason for the success of this symposium.

September 2005

Tom Conte
Nacho Navarro
Wen-mei W. Hwu
Mateo Valero

Organization

Executive Committee

General Chairs	Tom Conte (NC State University, USA) Nacho Navarro (UPC, Spain)
Program Committee Chairs	Wen-mei W. Hwu (UIUC, USA) Mateo Valero (UPC Barcelona, Spain)
Publicity Chair	Sally McKee (Cornell University, USA)
Publication Chair	Theo Ungerer (University of Augsburg, Germany)
Local Arrangements Chairs	Eduard Ayguade (UPC, Spain) Josep Llosa (UPC, Spain)
Registration/Finance Chair	Pilar Armas (UPC, Spain)
Web Chair	Michiel Ronsse (Ghent University, Belgium)

Program Committee

David August	Princeton University, USA
David Bernstein	IBM Haifa Research Lab, Israel
Mike O'Boyle	University of Edinburgh, UK
Brad Calder	University of California at San Diego, USA
Jesus Corbal	Intel Labs Barcelona, Spain
Alex Dean	NC State University, USA
Koen De Bosschere	Ghent University, Belgium
Jose Duato	UPV, Spain
Marc Duranton	Philips, France
Krisztian Flautner	ARM Ltd., Cambridge, UK
Jose Fortes	University of Florida, USA
Roberto Giorgi	University of Siena, Italy
Rajiv Gupta	University of Arizona, USA
Kazuki Joe	Nara Women's University, Japan
Manolis Katevenis	ICS, FORTH, Greece
Stefanos Kaxiras	University of Patras, Greece
Victor Malyshkin	Russian Academy of Sciences, Russia
William Mangione-Smith	UCLA, USA
Avi Mendelson	Intel, Israel
Enric Morancho	UPC, Spain
Jaime Moreno	IBM T.J. Watson Research Center, USA
Andreas Moshovos	University of Toronto, Canada
Trevor Mudge	University of Michigan, USA
Alex Nicolau	University of California, USA

Yale Patt	University of Texas at Austin, USA
Antonio Prete	University of Pisa, Italy
Alex Ramirez	UPC, Spain
Jim Smith	University of Wisconsin, USA
Per Stenström	Chalmers University, Sweden
Olivier Temam	INRIA Futurs, France
Theo Ungerer	University of Augsburg, Germany
Stamatis Vassiliadis	T.U. Delft, The Netherlands
Jingling Xue	University of New South Wales, Australia

Steering Committee

Anant Agarwal	MIT, USA
Koen De Bosschere	Ghent University, Belgium
Mike O'Boyle	University of Edinburgh, UK
Brad Calder	University of California, USA
Rajiv Gupta	University of Arizona, USA
Wen-mei W. Hwu	UIUC, USA
Josep Llosa	UPC, Spain
Margaret Martonosi	Princeton University, USA
Per Stenström	Chalmers University, Sweden
Olivier Teman	INRIA Futurs, France

Reviewers

Adrian Cristal	David H. Albonesi
Alexander V. Veidenbaum	Antonio Gonzalez
Alex Ramirez	David August
Jean-Loup Baer	David Bernstein
Angelos Bilas	Javier D. Bruguera
Brad Calder	Jose Maria Cela
Christine Eisenbeis	Christos Kozyrakis
Tom Conte	Doug Burger
Dan Connors	Alex Dean
Daniel A. Jimenez	Jose Duato
Michel Dubois	Marc Duranton
Eduard Ayguade	Enric Morancho
Esther Salamí	Krisztian Flautner
Jose Fortes	Jean-Luc Gaudiot
Roberto Giorgi	Rajiv Gupta
Wen-mei W. Hwu	Jaume Abella
Jose Manuel Garcia-Carrasco	Jaime Moreno
Kazuki Joe	Jorge Garcia
Jose M. Barcelo	Josep Llosa

Javier Verdú
Manolis G.H. Katevenis
Koen De Bosschere
Kunle Olukotun
William Mangione-Smith
José F. Martínez
Michael Francis O'Boyle
Andreas Moshovos
Walid A. Najjar
Oliverio J. Santana
Pradip Bose
Peter M.W. Knijnenburg
Michiel Ronsse
Ronny Ronen
Pascal Sainrat
Toshinori Sato
James Smith
Olivier Temam
Dean M. Tullsen
Stamatis Vassiliadis
Jingling Xue
Dionisios Pnevmatikatos
Daniele Mangano
Alessandro Bardine
Sandro Bartolini
Nidhi Aggarwal
Kyle Nesbit
Tejas Karkhanis
David Kaeli
Stefanos Kaxiras
Krste Asanovic
Victor Malyshkin
Mario Nemirovsky
Avi Mendelson
Ramon Beivide
Nacho J. Navarro
Alex Nicolau
Yale Patt
Per Stenström
Antonio Prete
Roger Espasa
Sarita V. Adve
Sandhya Dwarkadas
Andre Seznec
Jesus Sanchez
Josep Torrellas
Theo Ungerer
Valentin Puente
Yiannos Sazeides
George Apostolopoulos
Paolo Masci
Ida Savino
Paolo Bennati
Shiliang Hu
Jason Cantin
Wooseok Chang

Table of Contents

Invited Program

I Analysis and Evaluation Techniques

II Novel Memory and Interconnect Architectures

Keynote 1: Using EEMBC Benchmarks to Understand Processor Behavior in Embedded Applications

Markus Levy

President EEMBC

Abstract. Since its first introduction five years ago, benchmark software from the Embedded Microprocessor Benchmark Consortium (EEMBC) has gained critical mass as an industry standard for measuring the performance of embedded processors. From the beginning, EEMBC benchmarks have been distinguished by the application-specific representations of real processor tasks that they provide, and by the strict requirements for score certification before publication established by the Consortium. EEMBC is working to ensure its continued success by developing new benchmarks to reflect the evolving embedded market and to developing new ways to place the benchmark code in the hands of more engineers around the world.

This presentation begins with an introduction and overview of EEMBC's accomplishments and implementation strategy to date. From there, I will discuss the organization's future goals. Some of the hottest topics in the embedded industry include multicore systems and power conservation. Within EEMBC, work is now underway to develop multicore processor benchmarks that address both heterogeneous and homogenous processor implementations. A simultaneous effort is also underway, within a separate organization, to develop standards that will support the development of multicore platforms. These standards will initially address multicore debugging, on the chip and system level, and they will also address the issues of messaging and synchronization and how standards for scalable APIs could be leveraged and developed for embedded systems. On the power front, EEMBC is nearing the release of its specification for performing standardized power measurements on embedded platforms. Significantly, these measurements are performed while the platform is running the EEMBC benchmarks, thus helping to simultaneously characterize the performance and energy profile of a processor. But the real success of EEMBC's power measurements is that they work in a consistent manner, and, when released, will have been agreed to by the majority of vendors in the processor market.

This presentation will also describe other EEMBC benchmark projects currently under development, such as real-time automotive, office automation, and benchmarks to support VoIP applications. Using case studies, I will also demonstrate various methods for applying EEMBC to derive execution profiles and improve the designer's understanding of compiler and system-level design options.

Finally, the presentation will discuss how the adoption of these benchmarks within the embedded industry is expanding, with the implementa-

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 3–4, 2005.

tion of a new licensing program that makes it easier and more practical for system designers and researchers to gain access to the benchmark code.

Author Biography

Markus Levy is founder and president of EEMBC, the Embedded Microprocessor Benchmark Consortium. He is also technical editorial director for IQ Magazine, as well as chairman of the ARM Developer's Conference. Mr. Levy has more than nine years of experience working with EDN Magazine and Instat/MDR, and is a very seasoned editor and analyst with a proven record of processor and development tool analysis, article writing, and the delivery of countless technical seminars. Beginning in 1987, Mr. Levy worked for Intel Corporation as both a senior applications engineer and customer training specialist for Intel's microprocessor and flash memory products. While at Intel, he received several patents for his ideas related to flash memory architecture and usage as a disk drive alternative. Mr. Levy is also co-author of Designing with Flash Memory, the only technical book on this subject.

Keynote 2: The Chip-Multiprocessing Paradigm Shift: Opportunities and Challenges

Per Stenström

Chalmers University of Technology, Goteborg Sweden

Abstract. At a point in time when it is harder to harvest more instruction-level parallelism and to push the clock frequency to higher levels, industry has opted for integrating multiple processor cores on a chip. It is an attractive way of reducing the verification time by simply replicating moderately complex cores on a chip, but it introduces several challenges. The first challenge is to transform the processing power of multiple cores to application parallelism. The second challenge is to bridge the increasing speedgap between processor and memory by more elaborate on-chip memory hierarchies. Related to the second challenge is how to make more effective use of the limited bandwidth out of and into the chip.

In this talk I will elaborate on the opportunities that chipmultiprocessing offers but also the research issues that it introduces. Even if multiprocessing has been studied for more than two decades, the tight integration of cores and their on-chip memory subsystems opens up new unexplored terrains. I will discuss approaches to explore new forms of parallelism, approaches to bridge the processor/memory speedgap. In particular, I will focus on approaches to improve the way we utilize processor and memory resources. I will exemplify with approaches being studied in my own research as well as elsewhere. Finally, I will present an outlook of why I believe that parallel processing is one of the main hopes for the future of computer architecture and related fields, e.g. compilers.

Author Biography

Per Stenström is a professor of computer engineering at Chalmers University of Technology and adjunct professor and deputy dean of the IT University of Goteborg. His research interests are devoted to design principles for high-performance computer systems. He is an author of two textbooks and a hundred research publications. He is regularly serving program committees of major conferences in the computer architecture field. He has been an editor of IEEE Transaction on Computers, is an editor of Journal of Parallel and Distributed Computing, the IEEE TCCA Computer Architecture Letters, and editor-in-chief of the Journal of High-Performance Embedded Architectures and Compilation Techniques. He has served as General as well as Program Chair of the ACM/IEEE Int. Symposium on Computer Architecture. He is a member of ACM and senior member of the IEEE.

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, p. 5, 2005.

Software Defined Radio – A High Performance Embedded Challenge

Hyunseok Lee[1], Yuan Lin[1], Yoav Harel[1], Mark Woh[1], Scott Mahlke[1], Trevor Mudge[1], and Krisztian Flautner[2]

[1] Advanced Computer Architecture Laboratory,
Electrical Engineering and Computer Science Department,
University of Michigan, 1301 Beal Ave. Ann Arbor, MI 48105-2122
{leehzz, linyz, yoavh, mwoh, mahlke, tnm}@eecs.umich.edu
[2] ARM Ltd. 110 Fullbourn Road, Cambridge, UK CB1 9NJ
Krisztian.flautner@arm.com

Abstract. Wireless communication is one of the most computationally demanding workloads. It is performed by mobile terminals ("cell phones") and must be accomplished by a small battery powered system. An important goal of the wireless industry is to develop hardware platforms that can support multiple protocols implemented in software (software defined radio) to support seamless end-user service over a variety of wireless networks. An equally important goal is to provide higher and higher data rates. This paper focuses on a study of the wideband code division multiple access protocol, which is one of the dominant third generation wireless standards. We have chosen it as a representative protocol. We provide a detailed analysis of computation and processing requirements of the core algorithms along with the interactions between the components. The goal of this paper is to describe the computational characteristics of this protocol to the computer architecture community, and to provide a high-level analysis of the architectural implications to illustrate one of the protocols that would need to be accommodated in a programmable platform for software defined radio. The computation demands and power limitations of approximately 60 Gops and 100∼300 mW, place extremely challenging goals on such a system. Several of the key features of wideband code division multiple access protocol that can be exploited in the architecture include high degrees of vector and task parallelism, small memory footprints for both data and instructions, limited need for complex arithmetic functions such as multiplication, and a highly variable processing load that provides the opportunity to dynamically scale voltage and frequency.

1 Introduction

Hand held wireless devices are becoming pervasive. These devices represent a convergence of many disparate features, including wireless communication, real-time multimedia, and interactive applications, into a single platform. One of the most difficult challenges is to create the embedded computing systems for these

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 6–26, 2005.

devices that can sustain the needed performance levels, while at the same time operate within a highly constrained power budget to achieve satisfactory battery lifetimes. These computing systems need to be capable of supercomputer level performance levels with estimated performance levels of more than 60 Gops, while having a total power budget of about 100∼300 mW. The current generation of microprocessors and DSPs are not capable of meeting these performance and power requirements. The term "mobile supercomputer" has been used to describe such platforms [1].

In this work, we focus on the wireless communication aspect of hand held devices. Wireless communication is one of the most computationally intense workloads that is driven by the demand for higher and higher data rates. To support seamless service between various wireless networks, there is a high demand for a common hardware platform that can support multiple protocols implemented in software, generally referred to as software defined radio (SDR) [2]. A fundamental conflict exists when defining a computing platform for SDR, because performance, power, and flexibility are conflicting goals. At one extreme, which maximizes flexibility, are general purpose processors, where algorithms can be defined in high-level languages. At the other extreme, which maximizes performance and minimizes power, are application specific integrated circuits (ASIC). ASICs are hardwired solutions that offer almost no flexibility, but are the standard for current platforms.

Our goal, shared by others in the field, is to design and develop a programmable "mobile supercomputer" for SDR. It is first necessary to develop an understanding of the underlying requirements and computation characteristics of wireless protocols. The majority of the computation occurs at the physical layer of protocols, where the focus is signal processing. Traditionally, kernels corresponding to the major components, such as filters and decoders, are identified. Design alternatives for these are then evaluated on workloads targeted to their specific function. This approach has the advantage of dealing with a small amount of code. However, we have found that the interaction between tasks in SDR has a significant impact on the hardware architecture. This occurs because the physical layer is a combination of algorithms with different complexities and processing time requirements. For example, high computation tasks that run for a long period of time can often be disturbed by small tasks. Further, these small tasks have hard real-time deadlines, thus they must be given high priority. As a result, we believe it is necessary to explore the whole physical layer operation with a complete model.

From the many wireless protocols, we have selected the wideband code division multiple access (W-CDMA) protocol as a representative wireless workload to illustrate the operation of wireless communication systems. W-CDMA system is one of leading third generation wireless communication networks where the goal is multimedia service including video telephony on a wireless link [3]. W-CDMA improves over earlier cellular networks by increasing the data rate from 64 Kbps to 2 Mbps. Additionally, W-CDMA unifies a single service link for both voice and packet data, in contrast to previous generations which support only

one service. In order to study the requirements in more detail, we have developed a full C implementation of the W-CDMA physical layer to serve as the basis for our study. The implementation can be executed on a Linux workstation and thus studied with conventional architectural tools.

In this paper, we provide a detailed description and analysis of the W-CDMA physical layer. The fundamental computation patterns and processing time requirements of core algorithms are analyzed, along with the interactions between them. We also study the implications of the processing requirements on potential architectural decisions. One of the major keys to achieving the challenging power and performance goals is exploiting parallelism present in the computation, especially vector and task-level parallelism. This is balanced by a number of smaller real-time tasks that are more sequential in nature. Thus, it is important to consider both extremes and define an architecture capable of handling diverse types of processing.

The design of fully programmable architectures for W-CDMA is a difficult challenge faced by the industry. To date, no such design exists and thus serves as motivation for our analysis. Current DSP solutions, such as the TI TMS320C5XXX, have included specialized instructions, such as a compare-select instruction, designed specifically for wireless protocols. Further, there are many announced multiprocessor DSP systems that are designed specially for SDR. Some examples include the Sandblaster processor [4], the MorphoSys processor [5], and the 3plus1 processor [6]. However, none has yet to provide a fully programmable solution that could be programmed for other protocols and that satisfies both real-time W-CDMA performance requirement as well as being competitive with ASICs in power consumption. Some of these solutions, like MorphoSys processors, are aimed at basestations, where the power requirements are less stringent. To meet the W-CDMA processing requirements, many of these programmable processors also require ASIC accelerators for the most computationally demanding portions of the protocol.

2 W-CDMA Protocol

The protocol stack of the W-CDMA system consists of several layers and each layer provides an unique function in the system. According to the computation characteristics, we can group the protocol layers into two parts: the physical layer and upper layers. The physical layer, which is placed at the bottom of the protocol stack, is responsible for signal transmission over an unreliable wireless link. To overcome noise that the environment introduces on the wireless link, the physical layer performs many computation intensive signal processing algorithms such as matched filtering and forward error correction. Meanwhile, upper layer protocols cover the control signaling required to operate within a wireless network. For example, the medium access control (MAC) layer provides a scheme for resolving resource contention between multiple terminals.

Although SDR encompasses all protocols layers, we only focus on the physical layer due to its computation and power importance compared to other layers.

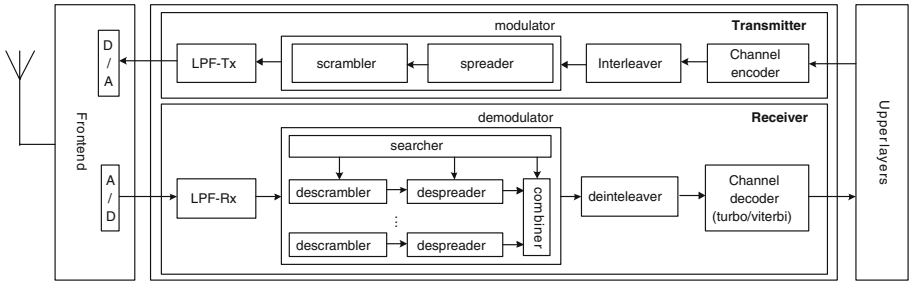


Fig. 1. High level block diagram of the physical layer operation of a W-CDMA terminal

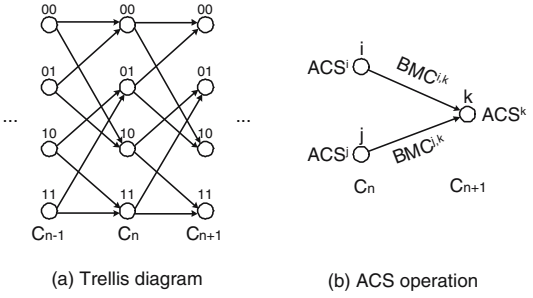
The operation of the physical layer is realized by both digital and analog circuits. Because the operation frequency of analog frontend circuits reaches GHz level, it is infeasible to replace the analog frontend circuits with programmable digital logic with current circuit technology. Thus, we further narrow down our focus to the physical layer that is realized with digital circuits. Figure 1 shows a high level block diagram of the physical layer operation of the W-CDMA terminal.

To aid in explanation, we define the following sets: a set of binary numbers with dual polarities, $\mathbf{B} = \{-1, 1\}$ [1]; a set of complex binary numbers, $\mathbf{B^C} = \{a + jb \mid a, b \in \mathbf{B}\}$; a set of m bit fixed point numbers, $\mathbf{I} = \{a \mid a, m$ are integers and, $-2^{m-1} < a \leq 2^{m-1}\}$; and a set of complex numbers represented by two m bit fixed point numbers, $\mathbf{I^C} = \{a + jb \mid a, b \in \mathbf{I}\}$.

Channel Encoder and Decoder. The role of a channel encoder and decoder is for error correction. The channel encoder in a transmitter adds systematic redundancy into the source information, and the channel decoder in a receiver corrects errors within the received information by exploiting the systematic regularity of the redundant information. The W-CDMA physical layer uses two kinds of channel coding schemes: **convolutional codes** [7] and **turbo codes** [8]. The detailed description of the channel codes for the W-CDMA physical layer is in [9]. The encoders for both codes are simple enough to be implemented with several flip-flops and exclusive OR gates. However, the decoders for these codes are highly complex because their operation is to find a maximum likely code sequence from the received noisy signal.

Among many possible methods, our implementations for the channel decoders are based on a soft output Viterbi algorithm (SOVA), because of its lower computational complexity and the fact that it only shows a slight performance degradation compared to other methods. The difference between a conventional Viterbi algorithm and the SOVA is the use of "soft" numbers in the SOVA. If a soft number is used, each bit plus noise in the received sequence is quantized as a fixed point number with higher precision (e.g. 4 bits). Although it requires more computation power and memory, the use of soft number is necessary in most practical W-CDMA receivers to provide high fidelity signal processing gain.

[1] In conventional binary notation $-1 \equiv 1$ and $\mathbf{1} \equiv 0$.



(a) Trellis diagram (b) ACS operation

Fig. 2. (a) The trellis diagram of a channel encoder comprising 2 flip-flops; (b) The ACS operation where the source nodes i, j are in the n-th column and the destination node k is in the $(n+1)$-th column

The operation of the SOVA is divided into three steps: branch metric calculation (BMC), add compare select (ACS), and trace back (TB). All steps of the Viterbi algorithm are based on a trellis diagram that consists of nodes representing the state of the channel encoder and arrows representing the state transition of the channel encoder. Figure 2(a) is an example of a trellis diagram. From a computation perspective, the BMC operation is equivalent to calculating the distance between two points, and so it can be expressed as follows:

$$BMC^{i,j} = distance(r_{ij}, o_{ij}) = abs(r_{ij} - o_{ij}) \tag{1}$$

where i is the source node; j is the destination node; $r_{ij} \in \mathbf{I}$ is the received signal; and $o_{ij} \in \mathbf{I}$ is the error free output of a channel encoder corresponding to the state transition from node i to j. The BMC operation on all nodes in a trellis diagram can be done in parallel because the inputs of the BMC operation on a node are independent of the result of the BMC operation on other nodes.

Assuming there exist two input transitions at node k from nodes i and j, the ACS operation on a node k can be represented by following equation:

$$ACS^k = min(ACS^i + BMC^{i,k}, ACS^j + BMC^{j,k}) \tag{2}$$

From the above equation, we can see the operation dependency between ACS operations: ACS^k depends on ACS^i and ACS^j. Therefore, the ACS operations of all nodes can not be done in parallel. However, nodes i, j and k in the above equation additionally have the following relation:

$$\text{if } i, j \in C_n, \text{then } k \in C_{n+1} \tag{3}$$

where C_n is a set of nodes in the n-th column of a trellis diagram (see Figure 2(a)). In other words, the ACS operations of the nodes in the $(n+1)$-th column can be done after the operations on the n-th column. Thus, at least the ACS operations of a column of a trellis diagram can be done in parallel.

The TB operation yields the most probable bit sequence which was transmitted by the transmitter based on the results for the BMC and ACS operation.

Because the TB operation is similar to transversing a linked list, the operation is inherently sequential.

This whole set of steps are usually referred to as convolutional coding. If turbo coding is required, further operations are needed. The turbo decoder is based on the repeated application of 2 concatenated SOVA decoders. The output of each SOVA is interleaved, and then feed into the other SOVA. The number of iterations varies according to channel conditions. Under good conditions, early termination is possible. Because of the data dependency between the iterations, parallelization of the iterations of the turbo decoder is not possible.

The parallelism available on the channel decoders is related to the number of flip-flops used at the channel encoders because it determines the length of columns in a corresponding trellis diagram. In the W-CDMA physical layer, the convolutional encoder comprises 8 flip-flops, so the length of the corresponding column vector is $2^8 = 256$; and the turbo encoder uses 3 flip-flops so the length of the column vector is $2^3 = 8$. In addition, the number of columns in a trellis diagram is also determined by the number of flip-flops. It has been shown that $5(n+1)$ columns in a trellis diagram is sufficient for reliable decoding where n is the number of flip-flops in a channel encoder [10]. Therefore, the SOVA requires 45 $(= 5 \times (8+1))$ columns in the trellis diagram for the convolutional code, and 20 $(= 5 \times (3+1))$ columns for the turbo code. Because all BMC operations can be done in parallel, the maximum number of parallel BMC operation is 11520 $(= 256 \times 45)$ for the convolutional code and 160 $(= 8 \times 20)$ for the turbo code. Because the ACS operations of one column can be done in parallel, the maximum number of parallel ACS operation is 256 for the convolutional code and 8 for the turbo code. In order to increase the parallelism of the turbo decoder, it is possible to decode multiple trellis diagrams simultaneously. This is known as the sliding window technique.

Interleaver and Deinterleaver. The interleaver and deinterleaver are used to overcome severe signal attenuation within a short time interval. In a wireless channel, an abrupt signal strength drop occurs very frequently. The interleaver in a transmitter randomizes the sequence of source information, and then the deinterleaver in a receiver recovers the original sequence by reordering. These operations scatter errors that occur within a short time interval over a longer time interval to reduce signal strength variation, and thus bit error rate, under the same channel conditions. Due to the randomness of the interleaving pattern, it is difficult to parallelize their operations without complex hardware support.

Modulator and Demodulator. Modulation maps source information onto signal waveforms so that they carry source information over wireless links most efficiently. Demodulation extracts that information from the received signal. In the W-CDMA physical layer, two classes of codes are deployed: channelization codes and scrambling codes.

A channelization code is used for multiplexing multiple source streams into one physical channel. In the transmitter, the procedure to multiply a channeliza-

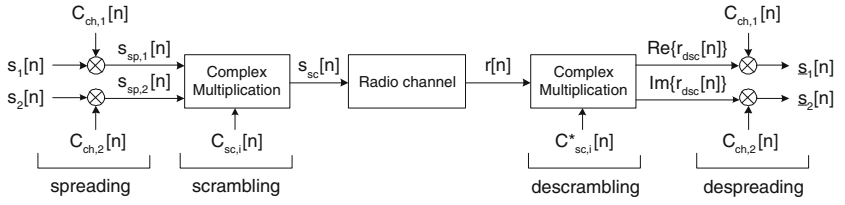


Fig. 3. Spreading/despreading and scrambling/descrambling operations with ignoring the operation of LPFs and analog frontend circuits

tion code with a source sequence is called **spreading**, because it spreads out the energy of the source information over a wider frequency spectrum. The following equation shows the spreading operation:

$$s_{sp,1}[n] = \mathrm{C}_{ch,1}[n \bmod L_{ch}] \cdot s_1[\lfloor n/L_{ch} \rfloor] \tag{4}$$

$$s_{sp,2}[n] = \mathrm{C}_{ch,2}[n \bmod L_{ch}] \cdot s_2[\lfloor n/L_{ch} \rfloor] \tag{5}$$

where $s[n] \in \mathbf{B}$ is a source data sequence provided by the interleaver; $\mathrm{C}_{ch}[n] \in \mathbf{B}$ is a channelization code; L_{ch} is the length of the channelization code; *mod* is a modulo operation; and $\lfloor \cdot \rfloor$ is a floor function. **Despreading** at the receiver reconstructs the estimated source data sequences $\underline{s}_1[n]$ and $\underline{s}_2[n]$. This procedure is shown by the following equations:

$$\underline{s}_1[n] = \sum_{i=0}^{L_{ch}-1} \mathrm{C}_{ch,1}[i] \cdot Re\{r_{dsc}[n \cdot L_{ch} + i]\} \tag{6}$$

$$\underline{s}_2[n] = \sum_{i=0}^{L_{ch}-1} \mathrm{C}_{ch,2}[i] \cdot Im\{r_{dsc}[n \cdot L_{ch} + i]\} \tag{7}$$

where $r_{dsc}[n] \in \mathbf{I^C}$ are the input of despreader which is provided by the descrambler. In the W-CDMA system, L_{ch} is a power of two from 4 to 512. It varies dynamically according to the source data rate such that the data rate of the output of spreading is fixed at 3.84 Mbps.

The use of a scrambling code enables us to extract the signal of one terminal when several terminals are transmitting signals at the same. The operation of multiplying a scrambling code with the output of the spreader is **scrambling** that is described by the following equation:

$$s_{sc}[n] = \mathrm{C}_{sc}[n \bmod L_{sc}] \cdot \{s_{sp,1}[n] + j s_{sp,2}[n]\} \tag{8}$$

where $\mathrm{C}_{sc}[n] \in \mathbf{B^C}$ is a scrambling code; L_{sc} is the length of the scrambling code; and $s_{sc,1}[n],\ s_{sc,2}[n] \in \mathbf{B}$ are the inputs of a scrambler that is generated by the spreaders. The corresponding action done in the receiver is **descrambling** which is described as follows:

$$r_{dsc}[n] = \mathrm{C}^*_{sc}[n \bmod L_{sc}] \cdot r[n] \tag{9}$$

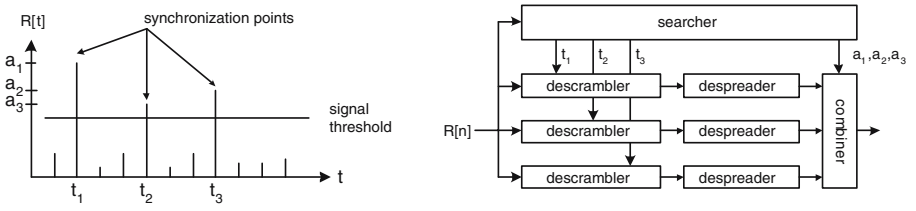


Fig. 4. (a) Example of cross correlation between the received signal and the scrambling code $C^*_{sc}[n]$ in a practical situation. Three synchronization points are detected by searcher at t_1, t_2, and t_3. (b) Structure of a rake receiver with three rake fingers. The operation times of each rake finger, t_1, t_2, and t_3, is set by searcher. The combiner aggregates the partial demodulation results of rake fingers.

where $r[n] \in \mathbf{I^C}$ is a received signal provided by low pass filters (LPF-Rx); and the * is a complex conjugate operation. In the W-CDMA physical layer, the data rate of all complex inputs and outputs of both scrambling and descrambling operations is fixed at 3.84 mega samples per second.

One assumption on the operation of the descrambler and despreader is that the receiver is perfectly synchronized with a transmitter. To achieve time synchronization, a receiver computes the cross correlations between the delayed version of a received signal $r[n-\tau] \in \mathbf{I^C}$, and a conjugated scrambling code sequence $\mathrm{C}^*_{sc}[n] \in \mathbf{B^C}$, by varying τ as follows:

$$R[\tau] = \sum_{i=0}^{L_{cor}-1} \mathrm{C}^*_{sc}[i] \cdot r[i+\tau], \text{ where } 0 \le \tau \le (L_s - 1) \tag{10}$$

where L_{cor} is the correlation length. In an ideal situation, $R[\tau]$ is maximized at the synchronization point because of the auto correlation property of the scrambling code: $\frac{1}{N}\sum_{i=0}^{N-1} C_{sc}[i] \cdot C^*_{sc}[i-\tau] = \delta[\tau]$. However, in a practical situation there are several correlation peaks because of multipath fading that an identical radio signal term arrives at a receiver multiple times with random attenuation and propagation delay. The multipath fading is caused by the reflection of the radio signal from objects placed on the signal propagation path. The **searcher** is an entity that finds synchronization points where the cross correlation $R[\tau]$ is greater than a predefined threshold level. Specifically, the operation of the searcher can be divided into four steps: 1) $R[\tau]$ calculation; 2) detection of local correlation peaks by calculating the derivative of $R[\tau]$; 3) filtering out high frequency noise terms from the local correlation peaks; and 4) global peaks detection by sorting the filtered local correlation peaks. The steps (1) and (2) have a high level of parallelism, whereas the steps (3) and (4) are difficult to parallelize. Details of the searcher can be found in [11]. The L_{cor} and L_s are important design parameters that significantly affect the workload of the W-CDMA physical layer. In our implementation, the L_{cor} and L_s are assumed to be 320 and 5120 correspondingly.

The receiver of the W-CDMA system descrambles and despreads a received signal at each of the synchronization points which are detected by the searcher, and then the partial demodulation results of the synchronization points are aggregated. This generation of partial demodulation results with proper delay compensation and the aggregation of these partial demodulation results mitigates the effect of the multipath fading. The aggregation of partial demodulation results is performed by a **combiner**. A receiver structure comprising the searcher, multiple demodulation paths, and the combiner is called **rake receiver** [12]. One demodulation path consisting of a descrambler and despreader is called as **rake finger**. The rake receiver is the most popular architecture used for CDMA terminals. In our implementation, we assumed the maximum number of rake fingers to be 12.

Low Pass Filter. A low pass filter (LPF) filters out signal terms that exist outside of an allowed frequency band in order to reduce the interference. Filtering can be represented by an inner product between an input data vector and a coefficient vector as follows:

$$y[n] = \sum_{i=0}^{L_{LPF}-1} c_i \cdot x[n-i] \tag{11}$$

where $x[n]$ is the input sequence to be filtered; the $c_i \in \mathbf{I}$ are the filter coefficients; and L_{LPF} is the number of filter coefficients.

There are two kinds of LPFs in the W-CDMA physical layer: LPF-Tx and LPF-Rx. Although the functionality of both filters are identical, the workload of both filters are different. The first difference is the size of operand. $x[n] \in \mathbf{B}$ in the LPF-Tx, but $x[n] \in \mathbf{I}$ in the LPF-Rx. The second difference is the number of LPF entities. At the LPF-Rx, there are two LPFs: one for the real part of the signal and the other for the imaginary part. However, the LPF-Tx consists of six LPFs (refer to the LPF-Rx in Figure 5). This structure is the result of an effort to reduce the amount of multiplication. A detailed explanation on the LPFs of the W-CDMA system can be found in [13]. In our implementation, L_{LPF} is 65.

Power Control. In general, CDMA systems adaptively control transmission power so that signals can be sent over the wireless channel at a minimum power level while satisfying a target bit error rate. The random variation of radio channel characteristics requires a feedback loop to control the strength of a transmitted signal. A transmitter sends reference signals called pilots, then a receiver sends back power control commands according to the quality of received pilot signals. To evaluate signal quality, it is necessary to fully demodulate the pilot signal in realtime. This sets a hard realtime requirement in the LPF, spreader/despreader, scrambler/descrambler and combiner. In the W-CDMA physical layer, the frequency of the power control operation is about 1.5KHz – every 0.67 msec.

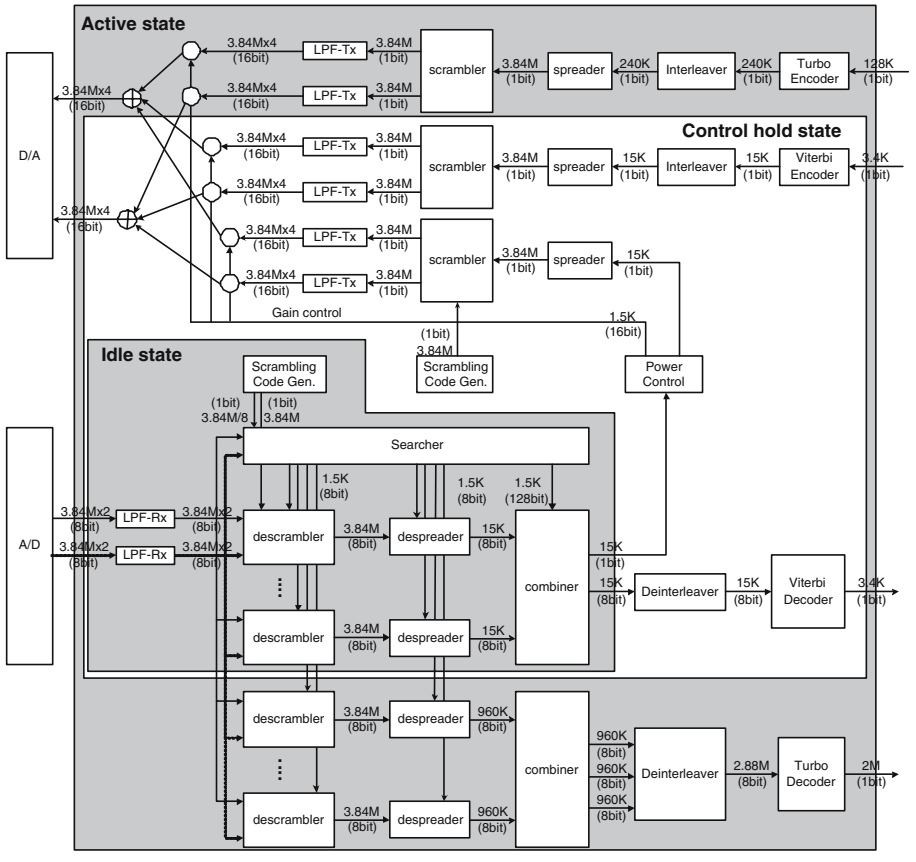


Fig. 5. Detailed block diagram of the W-CDMA physical layer providing the packet service described in Table 1

Operation State. From the view point of processor activity, it is possible to divide the operation of a W-CDMA terminal into three states[2]: idle, control hold, and active state. In the idle state a wireless terminal does not provide any application service to the user. However, even in this state, a terminal must be ready to respond to control commands from basestations. Although only simple tasks are performed in the idle state, the power consumed in this state is significant because a terminal spends most of its time in this state. In the

[2] In the W-CDMA standard, there are five radio resource control (RRC) states; camping on a UTRAN cell, ura_PCH, ura_FACH, cell_FACH, and cell_DCH states [14]. These RRC states are defined from the perspective of radio resource management. We redefine the operation state of the W-CDMA terminal according to processor activity. In our definition, the camping on a UTRAN cell, ura_PCH, ura_FACH, and cell_FACH states are grouped into the idle state. The cell_DCH state is divided into the control hold and active state.

active state, a terminal transmits and receives user data. It is the most heavily loaded operation state, because all function blocks are active. The control hold state is defined to represent the operation of a terminal during short idle periods between packet bursts. In the control hold state, a terminal maintains a low bandwidth control connection with basestations for fast transition to the active state when packets arrive.

3 Workload Analysis

Terminal Operation Conditions. Before going into the detailed workload analysis, we need to clarify the operation conditions of the W-CDMA terminal, because the workload of the W-CDMA physical layer is affected by several things: 1) operation state; 2) application type; and 3) radio channel status. Because the operation state significantly affects the hardware for the W-CDMA physical layer, we will analyze the variation in workload for all operation states. However, we limit the application type and radio channel condition.

Generally, we can classify application services into two categories: circuit service and packet service. Circuit service is a constant data rate service such as a voice call. Packet service is a variable data rate service such as internet access. Because the packet arrival pattern of the packet service demands a more complex resource management scheme, we select the packet service as our representative service. In addition, we further assume an asymmetric packet service that consists of a 2 Mbps link in the direction from basestation to terminal and a 128 Kbps link in the reverse direction. The asymmetric channel assumption matches the behavior of most packet services, for instance web browsing. For control signaling, the packet service additionally has a bidirectional signaling link with a 3.4 Kbps data rate.

The workload of a W-CDMA terminal is also varied by three radio channel conditions: 1) the number of basestations that communicate with a terminal at the same time; 2) the number of correlation peaks resulted in by the multipath fading; and 3) the quality of received signal. We assume 3 basestations, and 4 correlation peaks from the signal of a basestation. Thus, the W-CDMA terminal activates a total of 12 ($= 3 \times 4$) rake fingers. Because he quality of the received signal has a direct impact on the number of iterations of the turbo decoder, we assumed the quality of the received signal is set by the power control such that the average number of turbo decoder iterations is 3 per frame.

System Block Diagram. Figure 5 shows a detailed block diagram of the W-CDMA physical layer. It explains which algorithms participate in the action of each operation state. In the idle state, a subset of the reception path, the LPF-Rx and rake receiver, is active. In the control hold state, a bidirectional 3.4 Kbps signaling link is established with the basestations. Thus, a terminal activates both transmission and reception paths including the convolutional encoder/Viterbi decoder, LPF-Rx/Tx, modulator/demoulator, and power control. In the active state, a terminal additionally establishes a bidirectional high speed

Table 1. Assumed operation conditions of a W-CDMA terminal for workload analysis

Representation service	Service type	Packet service
	Data link	2 Mbps / 128 Kbps
	Signaling link	3.4 Kbps bidirectional
Channel condition	# of basestations	3
	# of rake fingers	12
	# of average turbo iterations	3

data link as shown in Table 1. Thus, the turbo encoder/decoder participate in the active state operation of a terminal.

Figure 5 also describes the interface between the algorithms of that make up W-CDMA. The number at the top of each arrow represents the number of samples per second, and that at the bottom represents the size of a sample. From these numbers, we can derive the amount of traffic between the algorithms. The size of most data in the transmission path is 1 bit, but it is 8 or 16 bit in the reception path because the channel decoders use soft numbers as explained previously. From the diagram, we can see that the data rate is abruptly changed by the spreader and despreader. In the transmission path, the data rate is up-converted from kilo sample per second to mega samples per seconds after the spreading operation. The reception path exhibits the reverse.

Processing Time. Table 2 shows that the W-CDMA physical layer is a mixture of algorithms with various processing time requirements. The * notation in the table indicates that the corresponding parameter is a design parameter. Other parameters in the table are explicitly specified by the W-CDMA standard. The processing time shown in the second and fourth columns is the allocated time

Table 2. Processing time requirements of the W-CDMA physical layer

	Active/Control hold state		**Idle state**
	Processing Time(ms)	**Execution Freq.(Hz)**	**Processing Time(ms)**
Searcher	Fixed(5*)	50*	Fixed(40*)
Interleaver/Deinterleaver	Fixed(10)	100	-
Convolutional encoder	Fixed(10/20/40)	Variable	-
Viterbi decoder			
Turbo encoder			
Turbo decoder	Variable(10∼50*)		
Scrambler/Descrambler	Fixed(0.67)	1500	Fixed(0.67)
Spreader/Despreader			
LPF-Rx			
LPF-Tx			-
Power control			

for one call to each algorithm. The task frequency in the third column is the number of times each algorithm is called within a second.

We assume that the searcher is executed every 20 msec because a radio channel can be considered as unchanging during this interval. The scrambler, spreader, and LPF have periodic and very strict processing time requirements, because they participate in the power control action. The convolutional code is mainly used for the circuit service with a constant data rate, so the Viterbi decoder needs to fulfill its operation before the arrival of the next frame to avoid buffer overflow. The processing time of the Viterbi decoder can be configured with 10, 20, or 40 msec intervals according to how the service frame length is configured. Whereas the turbo code aims the packet service with a burst packet arrival pattern. By buffering of bursty packets, can relax its processing time constraint significantly. We assume that the processing time of the turbo decoder varies between 10∼50 msec according to the amount of buffered traffic. In the idle state, tasks have loose timing constraints, so the searcher operation is sequentially performed with minimal hardware and the task frequency is not a concern.

Workload Profile. The detailed workload profile of the W-CDMA physical layer is shown in Table 3. For this analysis, we compiled our W-CDMA benchmark with an Alpha gcc compiler, and executed it on the M5 architectural simulator [15]. We measured the instruction count that is required to finish each algorithm. Peak workload of each algorithm is achieved by dividing the instruction count by the tightest processing time requirement of each algorithm shown in Table 2.

Table 3. Peak workload profile of the W-CDMA physical layer and its variation according to the operation state

	Active		**Control Hold**		**Idle**	
	(MOPS)	%	**(MOPS)**	%	**(MOPS)**	%
Searcher	**26538.0**	**42.1**	**26358.0**	**58.4**	**3317.3**	**37.7**
Interleaver	2.2	0.0	2.2	0.0	-	-
Deinterleaver	0.2	0.0	0.2	0.0	-	-
Conv. encoder	0.0	0.0	0.0	0.0	-	-
Viterbi Decoder	200.0	0.3	200.0	0.4	-	-
Turbo encoder	0.0	0.0	0.0	0.0	-	-
Turbo decoder	**17500.0**	**27.8**	**0.0**	**0.0**	-	-
Scrambler	245.3	0.4	245.3	0.5	-	-
Descrambler	**2621.4**	**4.2**	**2621.4**	**5.8**	**889.2**	**10.1**
Spreader	297.5	0.5	297.5	0.7	-	0.0
Despreader	**3642.5**	**5.8**	**3642.5**	**8.0**	**607.1**	**6.9**
LPF-Rx	**3993.6**	**6.3**	**3993.6**	**8.8**	**3993.6**	**45.3**
LPF-Tx	**7897.2**	**12.6**	**7897.2**	**17.4**	-	-
Power control	0.0	0.0	0.0	0.0	-	-
Total	62937.0	-	45272.9	-	8807.2	-

The first thing to note in Table 3 is that the total workload varies according to the operation state change. The total workloads in the control hold and idle states are about 72% and 14% of that in the active state. Second, the workload profile also varies according to the operation state. In the active and control hold states, the searcher, turbo decoder, LPF-Tx are dominant. In the idle state, the searcher and LPF-Rx are dominant.

Table 4. Intrinsic computations in the W-CDMA physical layer

Operations	**Algorithms**	**Description**
Exclusive OR	Spreader, Scrambler	$z = x \oplus y$
Conditional Complement	Searcher, Descrambler Despreader, LPF-Tx	$z = s\ ?\ x : -x$
Multiplication	LPF-Rx	$z = c \cdot x$
Scalar reduction	Searcher, Despreader, LPF	$z = \sum_{i=0}^{N-1} x_i$
Vector permutation	Viterbi/Turbo decoder, Searcher	$z[n] = x[n + p_n]$
BMC	Viterbi/Turbo decoder	$z = abs(x_0 - x_1)$
ACS	Viterbi/Turbo decoder	$z = min(x_0 + c_0, x_1 + c_1)$

Intrinsic Computations. Major intrinsic operations in the W-CDMA physical layer operation is listed in Table 4. As we discussed in Section 2, many algorithms in the W-CDMA physical layer are based on multiplication operations. Because multiplication is a power consuming operation, it is advantageous to simplify this into operations. First, the multiplications in the spreader and scrambler can be simplified to an exclusive OR, because both operands are either 1 or -1. By mapping {1, -1} to {0, 1}, we can use the exclusive OR operation instead of multiplication. Second, the multiplications in the searcher, descrambler, despreader, and LPF-Tx can be simplified into conditional complement operations, because one operand of the multiplications in these algorithms is either -1 or 1, and the other operand is a fixed point number. However, the multiplication of the LPF-Rx cannot be simplified because both operands are fixed point numbers. The operands of the multiplications in Equations (8), (9), and (10) are complex numbers. We can treat these operations as integer multiplications, because of the following relation: $(a + jb)(c + jd) = (ac - bd) + j(bd + ad)$.

For the frequent inner product operations of the searcher, despreader, and LPFs, we need a scalar reduction operation to add up all elements in a vector. A vector permutation is also required for the channel decoders and searcher, because either output or operand vector needs to be permuted. In channel decoders, the core operations are the BMC and ACS.

Instruction Type Breakdown. Figure 6 shows the instruction type breakdown for the W-CDMA physical layer and their weighted average. To obtain these results, a terminal is in the active state with peak workload. Instructions are grouped into seven categories: add/sub; multiply/divide; logic; load;

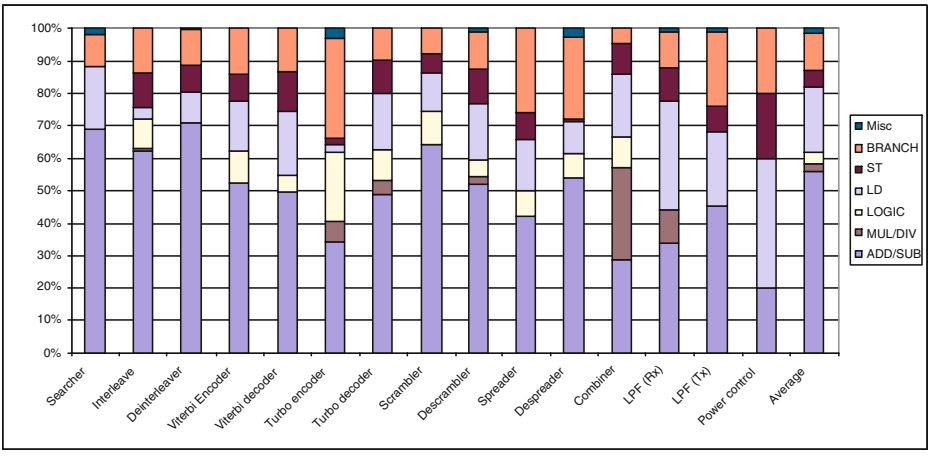


Fig. 6. Instruction type breakdown result

store; branches; and miscellaneous instructions. Because all algorithms are implemented with fixed point operations, there are no floating point instruction types shown here.

The first thing to notice is the high percentage of add/subtract instructions. They account for almost 50% of all instructions, with the searcher at about 70%. This is because the searcher's core operation is the inner product of two vectors, and the multiplication in an inner product can be simplified into a conditional complement. Furthermore, the complement operation is equivalent to a subtraction. Other hotspots, like the turbo decoder, also have a large number of addition operations. In the BMC and ACS of the turbo decoder, the additions are for calculating the distance between two points. In the TB of the turbo decoder, the additions come from pointer chasing address calculation.

The second thing to notice is the lack of multiplications/divisions (∼3% on average). This is because the multiplications of major algorithms are simplified into logical or arithmetic operations as discussed earlier. The multiplication of the combiner and turbo encoder is not a significant because their workload is very small as shown in Table 3. One exception is multiplications in the LPF-Rx. Figure 6 also shows that the number of load/store operations are significant. This is because most algorithms consist of loading two operands and storing the operation result.

Results also show that the portion of branch operations is about 10% on average. Most frequent branch patterns are loops with a fixed number of iterations corresponding to a vector size, and a conditional operation on vector variables. There are a few while or do-while loops, and most loops are 1 or 2 levels deep.

Parallelism. To meet the W-CDMA performance requirements in software, we must exploit the inherent algorithmic parallelism. Table 5 shows a breakdown of the available parallelism in the W-CDMA physical layer. We define data level

parallelism (DLP) as the maximum single instruction multiple data (SIMD) vector width and thread level parallelism (TLP) as the maximum number of different SIMD threads that can be executed in parallel. The second and third columns in the table are the ratio between the run time of the scalar code and the vector code. The fourth column represents maximum possible DLP. Because a vector operation needs two operands, we separately represent the bit width of two vector operands in the fifth column. The last column shows the TLP information.

Table 5. Parallelism available in the algorithms of the W-CDMA physical layer

		Scalar workload (%)	**Vector workload (%)**	**Vector width (elements)**	**Vector element width (bit)**	**Max concurrent Thread**
Searcher		3	97	320	1,8	5120
Interleaver		100	0	-	-	-
Deinterleaver		100	0	-	-	-
Viterbi encoder		60	40	8	1,1	1
Viterbi Decoder	BMC	1	99	256	8,8	45
	ACS	1	99	256	8,8	45
	TB	100	0	-	-	-
Turbo encoder		60	40	4	1,1	2
Turbo Decoder	BMC	1	99	16	8,8	20
	ACS	1	99	16	8,8	20
	TB	100	0	-	-	-
Scrambler		1	99	2560	1,1	1
Descrambler		1	99	2560	1,8	1
Spreader		100	0	-	-	-
Despreader		100	0	-	-	-
Combiner		100	0	-	-	-
LPF-Tx		1	99	65	1,16	6
LPF-Tx		1	99	65	8,8	2
Power Control		100	0	-	-	-

From Table 5, we can see that the searcher, LPF, scrambler, descrambler, and the BMC of the Viterbi decoder contain large amounts of the DLP and TLP. For the case of the scrambler and descrambler, it is possible to convert the DLP into TLP by subdividing large vectors into smaller ones. Although it is one of dominant workloads, the turbo decoder contains limited vector parallelism because the allowed maximum vector length of the ACS operation of the turbo decoder is 8.

There are also many unparallelizable algorithms in the W-CDMA physical layer. The interleaver, deinterleaver, spreader, despreader, and combiner operations have little DLP and TLP. Fortunately, the workload of these algorithms is not significant as show in Table 3. Therefore we can easily increase system throughput by exploiting the inherent parallelism shown in Table 5.

Table 6. Memory requirements of the algorithms of the W-CDMA physical layer

	Data memory (Kbyte)						**Inst. Memory** (Kbyte)
	I-buffer		**O-buffer**		**Scratch pad**		
	Kbyte	Mbps	Kbyte	Mbps	Kbyte	Mbps	
Searcher	**20.8**	**2.1**	**0.1**	**0.1**	**32.0**	**2654.3**	**3.1**
Interleaver	1.2	1.1	1.2	1.1	9.5	1.9	0.1
Deinterleaver	**26.1**	**5.2**	**26.1**	**5.2**	**8.7**	**5.2**	**0.1**
Viterbi Encoder	0.1	0.1	0.1	0.1	0.1	0.1	0.3
Viterbi Decoder	0.1	0.1	0.1	0.1	4.8	2.1	1.6
Turbo Encoder	2.6	4.0	7.8	12.0	0.1	2.0	1.6
Turbo Decoder	**61.5**	**96.0**	**2.6**	**4.0**	**6.4**	**25600.0**	**3.4**
Scrambler	0.7	15.4	0.7	15.4	0.7	15.4	0.5
Descrambler	**5.6**	**123.2**	**5.6**	**123.2**	**0.7**	**15.4**	**0.5**
Spreader	0.1	1.9	0.4	7.6	0.1	3.9	0.4
Despreader	0.4	7.6	0.1	1.9	0.1	3.9	0.3
Combiner	0.1	0.1	0.1	0.1	0.1	0.1	0.1
LPF-Tx	**0.3**	**7.6**	**10.3**	**245.8**	**0.1**	**1996.8**	**0.2**
LPF-Rx	**10.5**	**245.8**	**2.5**	**61.4**	**0.1**	**1996.8**	**0.2**
Power control	0.1	0.1	0.1	0.1	0.1	0.1	0.1

Memory Requirement. Because memory is one of the dominant power consuming elements in most systems, the analysis of the characteristics of memory is important. In general, there are two types of memory in a hardware system: data memory and instruction memory. Table 6 presents the data and instruction memory for all the algorithms in W-CDMA.

Columns 2∼7 in Table 6 show the size of the required data memory. The data memory is further divided into two categories: I/O buffer and scratch pad. The I/O buffer memory is used for buffering streams between algorithms. The scratch pad memory is temporary space needed for algorithm execution. We analyzed both size and access bandwidth of the data memory.

From the table, we can see that the W-CDMA algorithms require a small amount data memory, generally less than 64 Kbyte. In addition, we can see that the scratch pad memory is the most frequently accessed, especially in the searcher, turbo decoder, and LPFs. The access of I/O memory does not occupy a significant portion at the total memory access.

The last column of Table 6 shows the instruction memory size for each algorithm. For the analysis of the instruction memory size, we compiled our benchmark program on an Alpha processor. The average code size is less than 1 Kbyte and most kernels are below 0.5 Kbyte. This result is typical of many digital signal processing algorithms.

4 Architectural Implications

System Budget. The power budget allocated to baseband signal processing in commercial wireless terminals is typically between 100∼300 mW. Using this

Table 7. Estimation of system budgets for SDR applications and the achievable level of some typical commercial processors

	Power Efficiency (MOPS/mW)
Budget for SDR applications	210∼630
DSP: TI TMS320C55X [16][17]	∼40
Embedded Processor: AD ADSP-TS201 Tigersharc [18][19]	∼5
GPP: AMD Athelon MP [20][21]	∼0.003

power budget and the total peak workload data from Table 3, we calculated that the power efficiency of a processor for the W-CDMA physical layer must be between 210∼630 MOPS/mW. Because W-CDMA is one of the leading emerging wireless communication networks, we take this result as the system budget for typical SDR applications. In addition, we have calculated the maximum performance and energy efficiency of three conventional architectures: a digital signal processor (DSP), an embedded processor, and a general purpose processor (GPP). As shown in Table 7, these conventional architectures are a long way off of satisfying the requirements of SDR.

SIMD Processor. One key aspect for supporting the W-CDMA physical layer is to use an SIMD architecture. There are two main reasons why an SIMD architecture is beneficial. First, is that the W-CDMA physical layer contains a large amount of DLP as shown in Table 5. Second, the types of computations in tasks with high levels of DLP are mainly add/subtract operations. The absence of complex operations allows us to duplicate the SIMD functional units with minimal area and power overhead. Furthermore, we can expect significant power gain because a SIMD architecture can execute multiple data elements with one instruction decoding.

The intrinsic instructions shown in Table 4 can be used as a guide to designing the datapath and instruction set of the SIMD processor. In addition to the narrow data widths and simple operations, the datapath of the SIMD processor needs a vector status register that stores the status for each element of vector operations such as carry, zero and overflow. These vector status registers are useful for the realization of the conditional complement shown in Table 4. The datapath of the SIMD processor also needs a shuffle network to support vector permutation operations. The output of the arithmetic unit is shuffled before being stored in the register file. The hardware complexity of the shuffle network is exponentially proportional to the number of input nodes, which sets a limit on the width of the SIMD processor.

Scalar Processor. In addition to SIMD support, the W-CDMA physical layer requires scalar support because there are many small scalar algorithms in the W-CDMA physical layer as shown in Table 5. These scalar algorithms can be broken down into two main types: control and management tasks like the power control and rake receiver; and computation intensive DSP operations like the

TB of the turbo decoder. Most of the control operations have tight response time requirements, so there is a need for realtime interrupt support in the scalar processor. A conventional low power GPP such as an ARM processor is adequate for the scalar processor.

Multiprocessor. To further enhance the performance, a multiprocessor architecture is desirable if the TLP in the W-CDMA physical layer is to be exploited. Implementing a multiprocessor architecture raises many design questions: the granularity of the processing element (PE); the application task partitioning and mapping; the inter-process communication (IPC) mechanisms; and the heterogeneity of the PE.

One key factor which determines the efficiency of a multiprocessor architecture is the amount of IPC. Communication over an IPC network takes longer and dissipates more power than an internal memory structure. Because of this, the size of the PEs and the method of mapping and partitioning application tasks should be determined so as to minimize IPC. In addition, the choice of the IPC mechanism is important, because the communication pattern of the W-CDMA physical layer is point to point. This favors message passing, because copying of data from one PE to another is more power efficient.

Another method for enhancing performance is using heterogeneous PEs. It is possible to save chip area by implementing the multiplier only on a subset of PEs because multiplication is not used for all W-CDMA algorithms. Though heterogeneous PEs are desirable, homogeneous PE reduces development cost by reusing PE design.

Memory Hierarchy. In the memory hierarchy, certain choices optimizing for power is important. In the local memory of the PEs, the size of the memory is crucial for power efficiency, because, as shown in Table 6, there is very high internal memory traffic in the algorithms. Minimizing the size of these memories is necessary for more power efficient accesses. Also a global memory should be used because the W-CDMA physical layer requires the buffering of large bursty data traffic–input data traffic to the turbo decoder. This removes the need for large and power inefficient local memories.

For the W-CDMA physical layer, data and instruction caches are not essential. The memory access pattern of the W-CDMA physical layer is highly deterministic and exhibits very dense spacial locality, so it is possible to schedule memory access patterns in small sized memories. The advantages of cache free memories are low power consumption and a deterministic operation time. In DSP applications, a deterministic operation time is necessary for meeting timing requirements and easy system validation.

Power Management. Two power management challenges arise from the workload profile presented in Table 3: (1) Optimize the system for wide workload variations in the active and control hold states; (2) Minimize the power consumption in the idle state. Dynamic voltage scaling (DVS) and dynamic frequency scaling (DFS) are attractive ways to tackle the first challenge. The operation voltage and frequency can be dynamically adjusted according to the variation of wireless

channel conditions and user traffic. In order to minimize the power consumption in the idle state, we can selectively turn-off blocks which are not required in this state, such as the LPF-Tx and turbo decoder. In addition, the operation of the LPF-Rx and search must be optimized for power, because these tasks are the leading power consumers in the idle state.

5 Conclusion

Software defined radio presents a new challenge for the architecture community. A programmable SDR solution needs to offer supercomputer performance, while running on mobile devices with ultra low power consumption. In this paper, we have presented a complete W-CDMA system benchmark and an analysis of its computational characteristics. The benchmark includes realistic operating conditions, and algorithm configurations that are based on commercial W-CDMA implementations. From our study, W-CDMA shows very unique dynamic behavior characteristics. It has ultra high performance requirements, and dynamically changing real-time processing requirements. The algorithms are dominated by addition/subtraction, small memory footprints, and a high degree of data and task level parallelism. The results indicate that a programmable architecture needs SIMD as well as scalar support, can be optimized for narrow width operations, requires only small instruction and data memories, and can exploit dynamic voltage scaling to account for dynamic workload changes. Our benchmarks provide a useful evaluation for future architecture studies of SDR. In the future, we will include 802.11x protocols into our benchmark suite to gain further understanding of the programmability and computation requirements of wireless protocols.

References

1. Austin, T., et al.: Mobile Supercomputers. IEEE Computer **37** (2004) 82-84
2. Tuttlebee, W.: Software Defined Radio: Baseband Technology for 3G Handset and Basestations. 1st edn. John Wiley & Sons, New York (2002)
3. Holma, H., Toskala, A.: W-CDMA for UMTS: Radio Access for Third Generation Mobile Communications. John Wiley & Sons, New York (2000)
4. http://www.sandbridgetech.com/
5. http://gram.eng.uci.edu/morphosys/
6. http://www.3p1t.com/
7. Forney, G., D., Jr.: The Viterbi Algorithm. Proc. IEEE **61** (1973) 268-278
8. Berrou, C., Glavieux, A., Thitimjshima, P.: Near Shannon Limit Error Correcting Coding and Decoding: Turbo-Codes. ICC (1993)
9. 3GPP TS 25.211, Multiplexing and Channel Coding (FDD)
10. Parhi, K., Nishitani, T.: Digital Signal Processing for Multimedia Systems. 1st edn. Marcel Dekker, New York (1999)
11. Grayver, E., et al.: Design and VLSI Implementation for a WCDMA Multipath Searcher. IEEE Trans. on Vehicular Technology **54** (2005) 889-902
12. Rappaport, T.: Wireless Communications: Principles and Practice. IEEE Press, Piscataway (1996)

13. Berenguer, I., et al.: Efficient VLSI Design of a Pulse Shaping Filter and DAC interface for W-CDMA transmission. 16th IEEE International ASIC/SoC Conference (2003)
14. 3GPP TS 25.331: Radio Resource Control (RRC) Protocol Specification
15. Binkert, N., Hallnor, E., and Reinhardt, S.: Network-Oriented Full-System Simulation using M5. CAECW (2003)
16. http://www.bdti.com/procsum/tic55xx.htm
17. Dielissen, J., et al.: Power-Efficient Layered Turbo Decoder processor. DATE (2001)
18. http://www.analog.com/
19. Bursky, D.: DSPs Attack Throughput Needs with 600-MHz Clocks and eDRAM. Electronic Design **51** (2003)
20. http://www.amd.com/
21. Feng. W.: Honey, I Shrunk the Beowulf!. ICPP (2002)

A Practical Method for Quickly Evaluating Program Optimizations

Grigori Fursin[1,2], Albert Cohen[1], Michael O'Boyle[2], and Olivier Temam[1]

[1] ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud University, France
{grigori.fursin, albert.cohen, olivier.temam}@inria.fr
[2] Institute for Computing Systems Architecture, University of Edinburgh, UK
mob@inf.ed.ac.uk

Abstract. This article aims at making iterative optimization practical and usable by speeding up the evaluation of a large range of optimizations. Instead of using a full run to evaluate a single program optimization, we take advantage of periods of stable performance, called phases. For that purpose, we propose a low-overhead phase detection scheme geared toward fast optimization space pruning, using code instrumentation and versioning implemented in a production compiler.

Our approach is driven by simplicity and practicality. We show that a simple phase detection scheme can be sufficient for optimization space pruning. We also show it is possible to search for complex optimizations at run-time without resorting to sophisticated dynamic compilation frameworks. Beyond iterative optimization, our approach also enables one to quickly design self-tuned applications.

Considering 5 representative SpecFP2000 benchmarks, our approach speeds up iterative search for the best program optimizations by a factor of 32 to 962. Phase prediction is 99.4% accurate on average, with an overhead of only 2.6%. The resulting self-tuned implementations bring an average speed-up of 1.4.

1 Introduction

Recently, iterative optimization has become an increasingly popular approach for tackling the growing complexity of processor architectures. Bodin et al. [7] and Kisuki et al. [22] have initially demonstrated that exhaustively searching an optimization parameter space can bring performance improvements higher than the best existing static models, Cooper et al. [13] have provided additional evidence for finding best sequences of various compiler transformations. Since then, recent studies [34,19] demonstrate the potential of iterative optimization for a large range of optimization techniques.

Some studies show how iterative optimization can be used *in practice*, for instance, for tuning optimization parameters in libraries [38,6] or for building static models for compiler optimization parameters. Such models derive from the automatic discovery of the mapping function between key program characteristics and compiler optimization parameters; e.g., Stephenson et al. [32] successfully applied this approach to unrolling.

However, most other articles on iterative optimization take the same approach: several benchmarks are repeatedly executed with the same data set, a new optimization parameter (e.g., tile size, unrolling factor, inlining decision,...) being tested at each execution. So, while these studies demonstrate the *potential* for iterative optimization,

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 29–46, 2005.

few provide a *practical* approach for effectively applying iterative optimization. The issue at stake is: what do we need to do to make iterative optimization a reality? There are three main caveats to iterative optimization: quickly scanning a large search space, optimizing based on and across multiple data sets, and extending iterative optimization to complex composed optimizations beyond simple optimization parameter tuning.

In this article, we aim at the general goal of making iterative optimization a usable technique and especially focus on the first issue, i.e., how to speed up the scanning of a large optimization space. As iterative optimization moves beyond simple parameter tuning to composition of multiple transformations [19,26,11] (the third issue mentioned above), this search space can become potentially huge, calling for faster evaluation techniques. There are two possible ways to speeding up the search space scanning: search more smartly by exploring points with the highest potential using genetic algorithms and machine learning techniques [12,13,35,33,3,25,20,32], or scan more points within the same amount of time. Up to now, speeding up the search has mostly focused on the former approach, while this article is focused on the latter one.

The principle of our approach is to improve the efficiency of iterative optimization by taking advantage of program *performance stability* at run-time. There is ample evidence that many programs exhibit phases [30,23], i.e., program trace intervals of several millions instructions where performance is similar. What is the point of waiting for the end of the execution in order to evaluate an optimization decision (e.g., evaluating a tiling or unrolling factor, or a given composition of transformations) if the program performance is stable within phases or the whole execution? One could take advantage of phase intervals with the same performance to evaluate a different optimization option at each interval. As in standard iterative optimization, many options are evaluated, except that multiple options are evaluated within the same run.

The main assets of our approach over previous techniques are simplicity and practicality. We show that, for many benchmarks, a low-overhead performance stability/phase detection scheme is sufficient for optimization space pruning. We also show that it is possible to search (even complex) optimizations at run-time without resorting to sophisticated dynamic optimization/recompilation frameworks. Beyond iterative optimization, our approach also enables one to quickly design self-tuned applications, significantly easier than current manually tuned libraries.

Phase detection and optimization evaluation are respectively implemented using code instrumentation and versioning within the EKOPath compiler. Considering 5 self-tuned SpecFP2000 benchmarks, our space pruning approach speeds up iterative search by a factor of 32 to 962, with a 99.4% accurate phase prediction and a 2.6% performance overhead on average; we achieve speedups ranging from 1.10 to 1.72.

The paper is structured as follows. Section 2 provides the motivation, showing how our technique can speedup iterative optimization, and including a brief description of how it may be applied in different contexts. Section 3 describes our novel approach to runtime program stability detection. This is followed in Section 4 by a description of our dynamic transformation evaluation technique. Section 5 describes the results of applying these techniques to well known benchmarks and is followed in Section 6 by a brief survey of related work. Section 7 concludes the paper.

2 Motivation

This section provide a motivating example for our technique and outlines the ways in which it can be used in program optimization.

2.1 Example

Let us consider the `mgrid` SpecFP2000 benchmark. For the sake of simplicity, we have tested only 16 random combinations of traditional transformations, known to be efficient, on the two most time consuming subroutines `resid` and `psinv`. These transformations include loop fusion/fission, loop interchange, loop and register tiling, loop unrolling, prefetching. Since the original execution time of `mgrid` is 290 seconds (for the reference data set), a typical iterative approach for selecting the best optimization option would take approximately $290 \times 32 = 9280$ seconds (more than 2 hours). Moreover, all these tests are conducted with the same data set, which does not make much sense from a practical point of view.

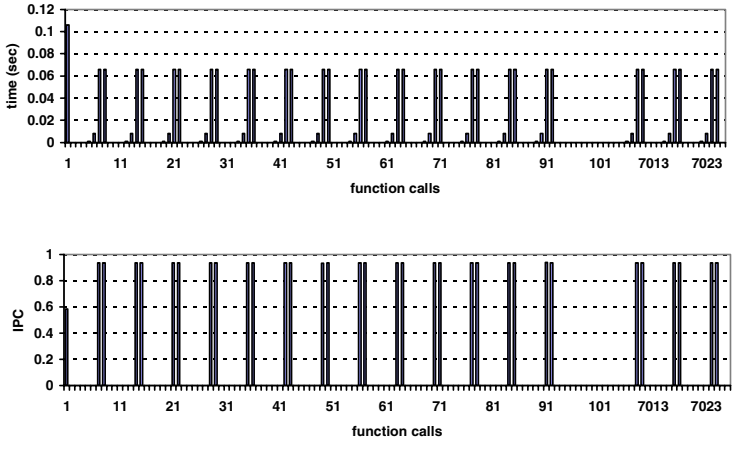


Fig. 1. Execution time and IPC for subroutine `resid` of benchmark `mgrid` across calls

However, considering the execution time of every call to the original subroutine `resid` in Figure 1, one notices fairly stable performance across pairs of consecutive calls with period 7.[1] Therefore, we propose to conduct most of these iterations at runtime, evaluating multiple versions during a single or a few runs of the application. The overall iterative program optimization scheme is depicted in Figure 2.

Practically, we insert all 16 different optimized versions of `resid` and `psinv` into the original code. As shown in the second box of Figure 2, each version is enclosed by calls to monitoring functions before and after the instrumented section. These timer

[1] Calls that take less than 0.01s are ignored to avoid startup or instrumentation overhead, therefore their IPC bars are not shown in this figure.

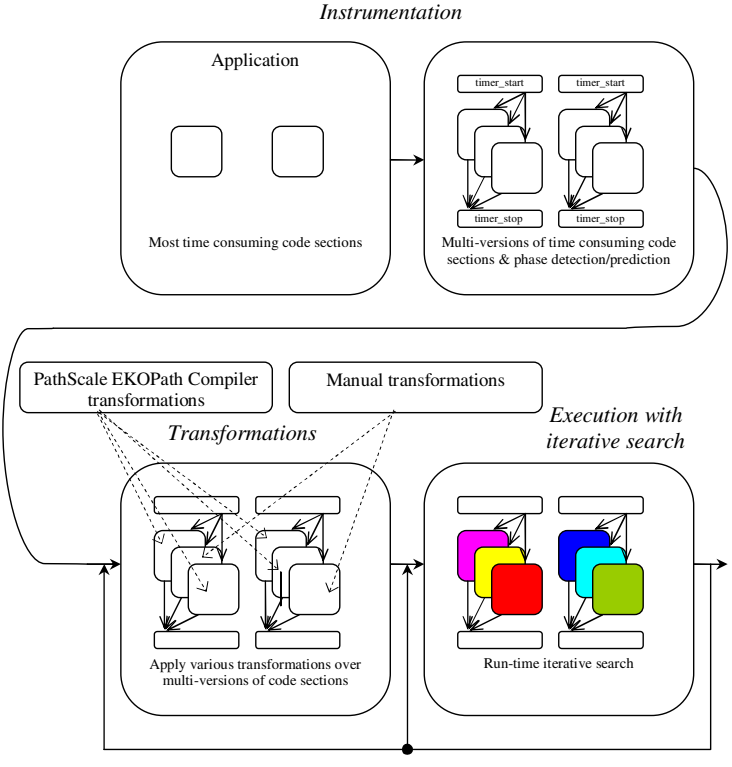


Fig. 2. Application instrumentation and multi-versioning for run-time iterative optimization

functions monitor the execution time and performance of any active subroutine version using the high-precision PAPI hardware counters library [8], allowing to switch at run-time among the different versions of this subroutine. This low-overhead instrumentation barely skews the program execution time (less than 1%) as shown in Figure 3.

If one run is not enough to optimize the application, it is possible to iterate on the multi-version program, the fourth box in Figure 2. Eventually, if new program transformations need to be evaluated, or when releasing an optimized application restricted

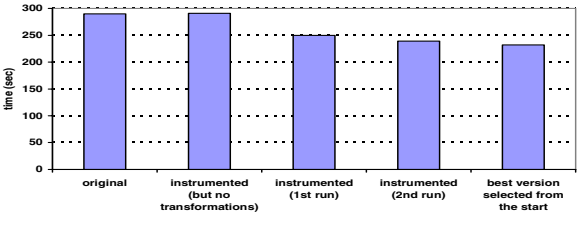


Fig. 3. Execution times for different versions of benchmark `mgrid`

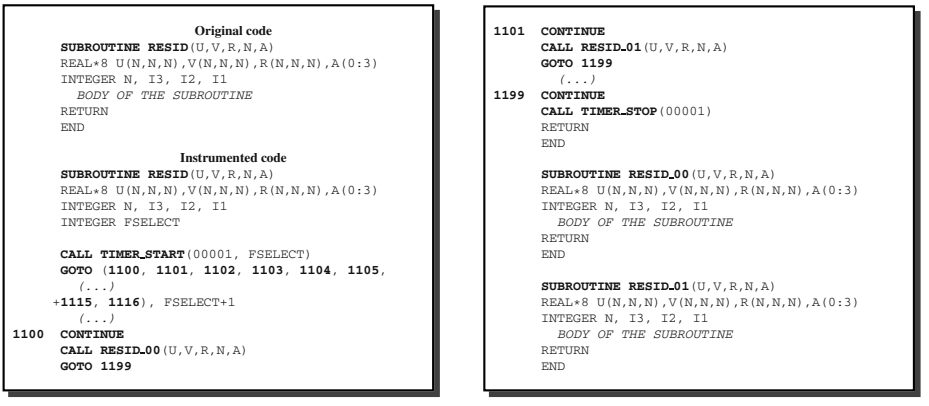


Fig. 4. Instrumentation example for subroutine `resid` of benchmark `mgrid`

to the most effective optimizations, one may also iterate back to apply a new set of transformations, the third box in Figure 2.

Figure 4 details the instrumentation and versioning scheme. Besides starting and stopping performance monitoring, `timer_start` and `timer_stop` have two more functions: `timer_stop` detects performance stability for consecutive or periodic executions of the selected section, using execution time and IPC; then `timer_start` predicts that performance will remain stable, in order to evaluate and compare new options. After stability is detected, `timer_start` redirects execution sequentially to the optimized versions of the original subroutine. When the currently evaluated version has exhibited stable performance for a few executions (2 in our case), we can measure its impact on performance *if* the phase did not change in the meantime. To validate this, the original code is executed again a few times (2 in our case to avoid transitional effects). In the same way all 16 versions are evaluated during program execution and the best one is selected at the end, as shown in Figure 5a.

Overall, evaluating all 16 optimization options for subroutine `resid` requires only 17 seconds instead of 9280 thus speeding up iterative search 546 times. Furthermore, since the best optimization has been found after only 6% of the code has been executed, the remainder of the execution uses the best optimization option and the overall `mgrid` execution time is improved by 13.7% all in one run (one data set) as shown in Figure 5b. The results containing original execution time, IPC and the corresponding best option which included loop blocking, unrolling and prefetching in our example, is saved in the database after the program execution. Therefore, during a second run with the same dataset (assuming standard across-runs iterative optimization), the best optimization option is selected immediately after the period is detected and the overall execution time is improved by 16.1% as shown in Figure 5c. If a different dataset is used and the behavior of the program changed, the new best option will be found for this context and saved into the database. Finally, the execution time of the non-instrumented code with the best version implemented from the start (no run-time convergence) brings almost the same performance of 17.2% as shown in Figure 5d and 3. The spikes on the graphs in Figure 5b,c are due to the periodic change in calling context of subroutine `resid`. At such

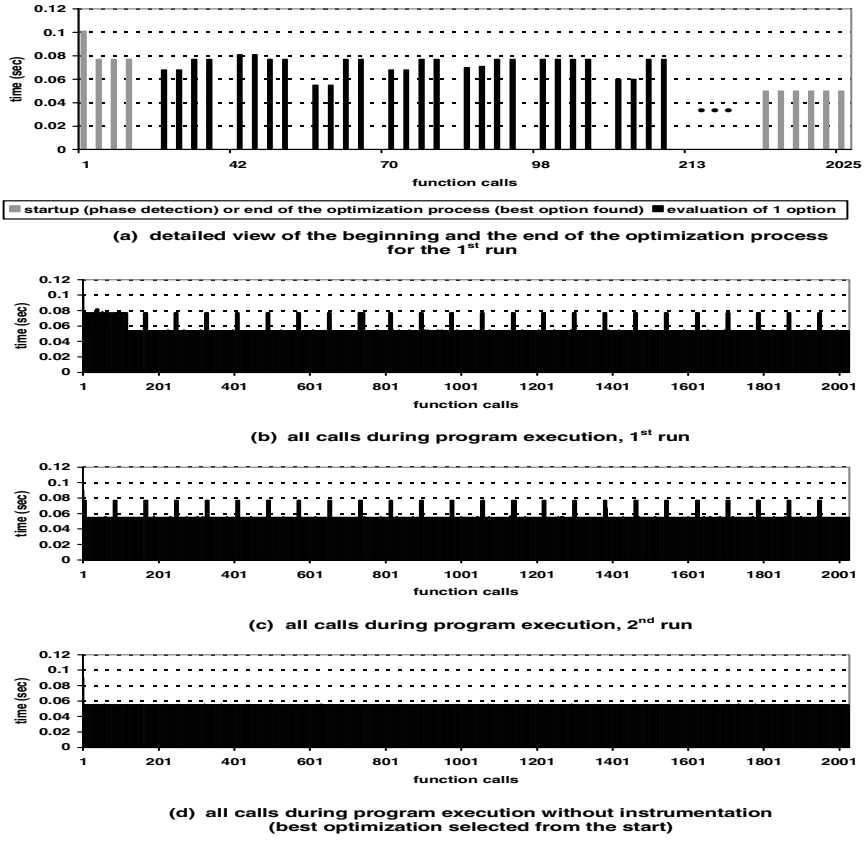


Fig. 5. Execution times for subroutine `resid` of benchmark `mgrid` during run-time optimization

change points, the phase detection mechanism produces a miss and starts executing *the original non-transformed version* of the subroutine, to quickly detect the continuation of this phase or the beginning of another one (new or with a known behavior).

2.2 Application Scenarios

The previous example illustrates the two main applications of our approach. The first one is iterative optimization, and the second is dynamic self-tuning code.

In the first case, each run — and each phase within this run — exercises multiple optimization options, including complex sequences of compiler or manual transformations. The phase analysis and optimization decisions are dumped into a database. This facility can be used for across-runs iterative optimization. There are two cases where this approach is practical. First, some applications may exhibit similar performance across multiple data sets, providing key parameters do not change (e.g., matrix dimensions do not change, but matrix values do); second, even when the performance of a

code section varies with the data set, it is likely that a few optimizations will be able to achieve good results for a large range of data sets. In both cases, run-time iterative optimization can speed up optimization space search: a selection of optimizations is evaluated during each run, progressively converging to the best optimization.

In the second case, our technique allows to create self-tuning programs which adjust to the current data set, within a production run. Assuming the optimized versions known to perform best (in general, for multiple benchmarks) have been progressively learned across previous runs and data sets, one can implement a self-tuning code by selecting only a few of those versions. Even if some of the selected versions perform poorly, they do not really affect overall execution time since convergence occurs quickly and most of the execution time is spent within the best ones.

3 Dynamic Stability Prediction

The two key difficulties with dynamic iterative optimization are how to evaluate multiple optimization options at run-time and when can they be evaluated. This section tackles the second problem by detecting and predicting stable regions of the program where optimizations may be evaluated.

3.1 Performance Stability and Phases

As mentioned in the introduction, multiple studies [30,27] have highlighted that programs exhibit *phases*, i.e., performance can remain stable for many millions instructions and performance patterns can recur within the program execution. Phase analysis is now extensively used for selecting sampling intervals in processor architecture simulation, such as in SimPoint [27]. More recently, phase-based analysis has been used to tune program power optimizations by dynamically adapting sizes of L1 and L2 caches [21].

For iterative optimization, phases mean that the performance of a given code section will remain stable for multiple consecutive or periodic executions of that code section. One can take advantage of this stability to compare the effect of multiple different optimization options. For instance, assuming one knows that two consecutive executions $E1$ and $E2$ of a code section will exhibit the same performance P, one can collect P in $E1$, apply a program transformation to the code section, and collect its performance P' in $E2$; by comparing P and P', one can decide if the program transformation is useful. Obviously, this comparison makes sense only if $E1$ and $E2$ exhibit the same baseline performance P, i.e., if $E1$ and $E2$ belong to the same phase. So, the key is to detect when phases occur, i.e., where are the regions with identical baseline performance. Also, IPC may not always be a sufficient performance metric, because some program transformations may increase or reduce the number of instructions, such as unrolling or scalar promotion. Therefore, we monitor not only performance stability but also execution time stability across calls, depending on the program transformations.

Figures 6 and 1 illustrate IPC and execution time stability of one representative subroutine for 5 SpecFP2000 benchmarks by showing variations across calls to the same subroutine. These benchmarks are selected to demonstrate various representative behavior for floating point programs. For instance, the `applu` subroutine has a stable

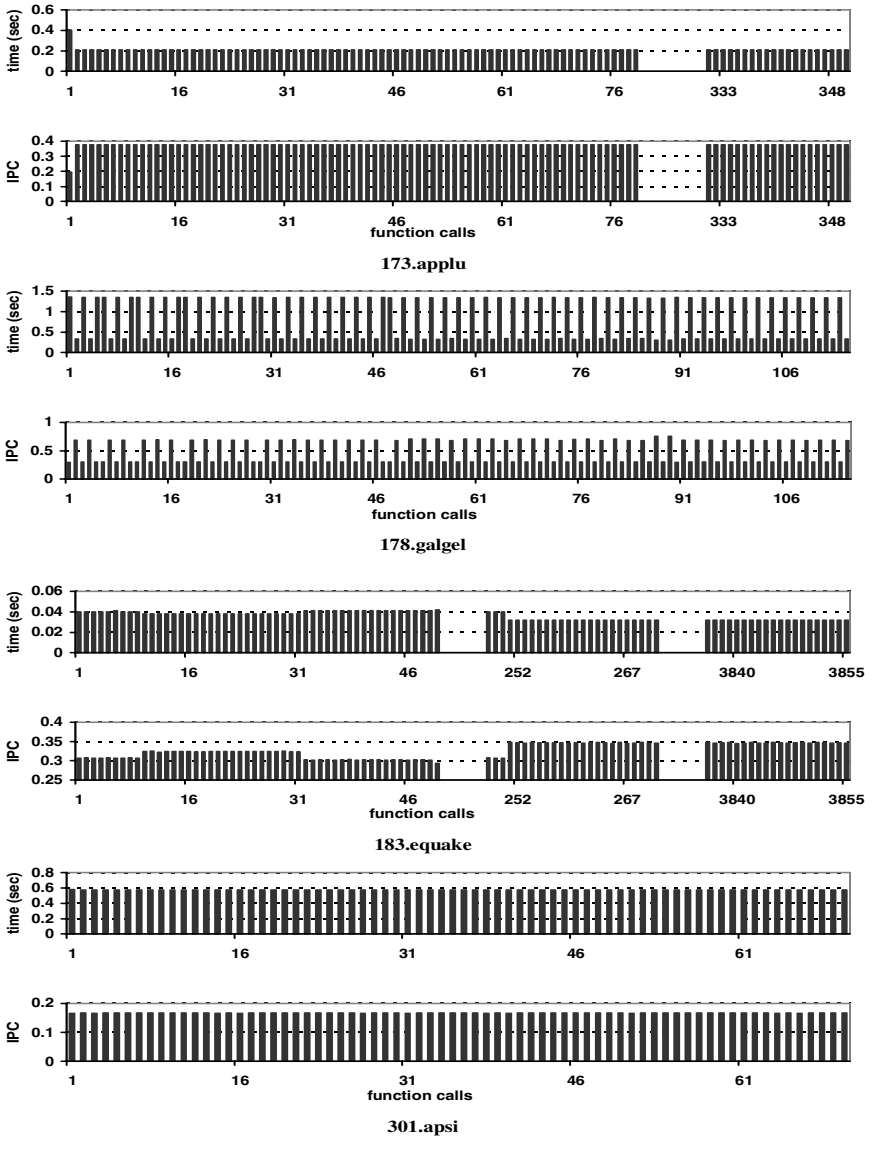


Fig. 6. Execution time and IPC for one representative subroutine per benchmark (across calls)

performance across all calls except for the first one; the `galgel` subroutine has periodic performance changes with 5 shifts during overall execution; the `equake` most time-consuming section exhibits unstable performance for 250 calls and then becomes stable; the `apsi` subroutine has a stable performance across all calls; finally, the `mgrid` subroutine exhibits periodic stable performance.

Detecting Stability. For the moment, we do not consider program transformations, and two instances are compared solely using IPC. We then simply define stability by 3 consecutive of periodic code section execution instances with the same IPC. Naturally, this stability characterization is speculative, the 4th instance performance may vary, but in practice as graphs in Figure 6 suggest, stability regions are long and regular enough so that the probability of incorrect stability detections (miss rate) is fairly low.

Note however, that the occurrence of a phase (an execution instance with a given performance) is only detected *after* it has occurred, i.e., when the counter value is collected and the code section instance already executed. If changes in the calling context occur faster than the evaluation of a new optimization option, there may not be long enough consecutive executions with stable performance to measure the impact of the optimization. Therefore, it is not sufficient to *react* to phase changes: within a phase, it is necessary to detect the *length of consecutive regions* of stable performance and to *predict* their occurrence. Fortunately, Figure 6 shows that phases tend to recur regularly, especially in scientific applications which usually have simple control flow behavior, which is further confirmed by other broader experiments [30]

To predict the occurrence of regular phases, for each instrumented code section, we store the performance measurement along with the number of calls exhibiting the same performance (phase length) in a Phase Detection and Prediction Table (PDPT) as shown in Figure 7. If a performance variation occured, we check the table to see if a phase with such behavior already occurred, and if so, we also record the distance (in number of calls) since it occurred. At the third occurrence of the same performance behavior, we conclude the phase becomes stable and recurs regularly, i.e., with a fixed period, and we can predict its next occurrence. Then, program transformations are applied to the code section, and the performance effects are only compared within the same

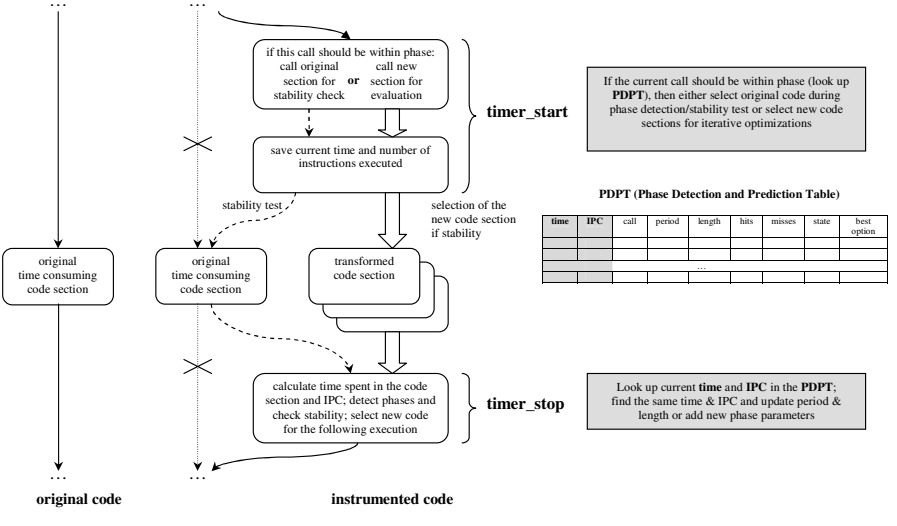


Fig. 7. Code instrumentation for run-time adaptive iterative optimizations

Table 1. Number of phases, hits and misses per code section for each application

Application	Code sections	Phases	Hits	Misses	Miss rate
mgrid	a	1	1924	27	0.014
	b	1	998	1	0.001
applu	a	1	348	0	0
	b	2	349	0	0
	c	2	349	0	0
	d	1	350	0	0
	e	1	350	0	0
galgel	a	2	86	12	0.140
	b	2	83	14	0.169
equake	a	2	3853	1	0.000
apsi	a	1	69	0	0
	b	1	69	0	0
	c	1	69	0	0
	d	1	69	0	0
	e	1	70	0	0
	f	1	69	0	0

phase, i.e., for the same baseline performance, avoiding to reset the search each time the phase changes. The length parameter indicates when the phase will change. Thus, the transformation space is searched independently for all phases of a code section. This property has the added benefit of allowing per-phase optimization, i.e., converging towards different optimizations for different phases of the same code section.

Table 1 shows how the phase prediction scheme performs. Due to the high regularity of scientific applications, our simple phase prediction scheme has a miss rate lower than 1.4% in most of the cases, except for `galgel` which exhibits miss rates of 14% and 17% for two time-consuming subroutines. Also, note that we assumed two performance measurements were identical provided they differ by less than a threshold determined by observed measurement error, of the order of 2% with our experimental environment.

3.2 Compiler Instrumentation

Since, program transformations target specific code sections, phase detection should target code sections rather than the whole program. In order to monitor code sections performance, we instrument a code section, e.g., a loop nest or a function, with performance counter calls from the hardware counters PAPI library. Figure 4 shows an example instrumentation at the subroutine/function level for `mgrid` (Fortran 77) and its `resid` subroutine. Figure 7 shows the details of our instrumentation.

Each instrumented code section gets a unique identifier, and before and after each section, monitoring routines `timer_start` and `timer_stop` are called. These routines record the number of cycles and number of instructions to compute the IPC (the first argument is the unique identifier of the section). At the same time, `timer_stop` detects phases and stability, and `timer_start` decides which optimization option

should be evaluated next and returns variable `FSELECT` to branch to the appropriate optimization option (versioning), see the `GOTO` statement.

Instrumentation is currently applied before and after the call functions and the outer loops of all loop nests with depth 2 or more (though the approach is naturally useful for the most time-consuming loop nests and functions only). Note that instrumented loop nests can themselves contain subroutine calls to evaluate inlining; however we forbid nested instrumentations, so we systematically remove outer instrumentations if nested calls correspond to loop nests.

4 Evaluating Optimizations

Once a stable period has been detected we need a mechanism to evaluate program transformations and evaluate their worth.

4.1 Comparing Optimization Options

As soon as performance stability is observed, the evaluation of optimization options starts. A new optimization option is said to be evaluated only after 2 consecutive executions with the same performance. The main issue is to combine the detection of phases with the evaluation of optimizations, because, if the phase detection scheme does not predict that a new phase starts, baseline performance will change, and we would not know whether performance variations are due to the optimization option being evaluated or to a new phase.

In order to verify the prediction, the instrumentation routine periodically checks whether baseline performance has changed (and in the process, it monitors the occurrence of new phases). After any optimization option evaluation, i.e., after 2 consecutive executions of the optimized version with the same performance, the code switches back to the original code section for two additional iterations. The first iteration is ignored to avoid transition effects because it can be argued that the previous optimized version of the code section can have performance side-effects that would skew the performance evaluation of the next iteration (the original code section). However, we did not find empirical evidence of such side-effects; most likely because code sections have to be long enough that instrumentation and start-up induces only a negligible overhead. If the performance of the second iteration is similar to the initial baseline performance, the effect of the current option is validated and further optimization options evaluation resumes. Therefore, evaluating an optimization option requires at least 4 executions of a given code section (2 for detecting optimization performance stability and 2 for checking baseline performance). For example, see the groups of black bars in Figure 5a of the motivation section (on benchmark `mgrid`). If the baseline performance is later found to differ, the optimization search for this other phase is restarted (or started if it is the first occurrence of that phase).[2]

Practically, the Phase Detection and Prediction Table (PDPT) shown in Figure 7 holds information about phases and their current state (detection and prediction), new

[2] Note that it is *restarted* not *reset*.

option evaluation or option validation (stability check). It also records the best option found for every phase.

4.2 Multiple Evaluations at Run-Time

Many optimizations are parameterized, e.g., tile size or unroll factor. However, in the context of run-time iterative optimization, whether changing a parameter just means changing a program variable (e.g., a parametric tile size), or changing the code structure (e.g., unroll factor) matters. The former type of optimization can be easily verified by updating the parameter variable. In order to accommodate the latter type of complex optimizations, we use versioning: we generate and optimize differently multiple versions of the same code section (usually a subroutine or a loop nest), plus the additional control/switching code driven by the monitoring routine as shown in Figures 7 and 4, using the EKOPath compiler.

The main drawback of versioning is obviously increased code size. While this issue matters for embedded applications, it should not be a serious inconvenience for desktop applications, provided the number of optimization options is not excessive. Considering only one subroutine or loop nest version will be active at any time, even the impact of versioning on instruction cache misses is limited. However, depending on what run-time adaptation is used for, the number of versions can vary greatly. If it is used for evaluating a large number of program transformations, including across runs, the greater the number of versions the better, and the only limitation is the code size increase. If it is used for creating self-adjusting codes that find the best option for the current run, it is best to limit the number of options, because if many options perform worse than the original version, the overall performance may either degrade or marginally improve. In our experiments, we limited the number of versions to 16.

This versioning scheme is simple but has practical benefits. Alongside the optimized versions generated by the compiler, the user can add subroutines or loop nests modified by hand, and either test them as is or combine them with compiler optimizations. User-suggested program transformations can often serve as starting points of the optimization search, and recent studies [15,34] highlight the key role played by well selected starting points, adding to the benefit of combined optimizations. Moreover, another study [11] suggests that iterative optimization should not be restricted to program transformation parameter tuning, but should expand to selecting program transformations themselves, beyond the strict composition order imposed by the compiler. Versioning is a simple approach for testing a variety of program transformations compositions.

5 Experiments

The goal of this article is to speedup the evaluation of optimization options, rather than to speedup programs themselves. Still, we later report program speedups to highlight that the run-time overhead has no significant impact on program performance, that the run-time performance analysis strategy is capable of selecting appropriate and efficient optimization options, and that it can easily accommodate both traditional compiler-generated program transformations and user-defined ad-hoc program transformations.

5.1 Methodology

Platforms and Tools. All experiments are conducted on an Intel Pentium 4 Northwood (ID9) Core at 2.4GHz (bus frequency of 533MHz), the L1 cache is 4-way 8KB, the L2 cache is 8-way 512KB, and 512MB of memory; the O/S is Linux SUSE 9.1. We use the latest PAPI hardware counter library [1] for program instrumentation and performance measurements. All programs are compiled with the open-source EKOPath 2.0 compiler and -Ofast flag [2], which, in average, performs similarly or better than the Intel 8.1 compiler for Linux.

Compiler-generated program transformations are applied using the EKOPath compiler. We have created an EKOPath API that triggers program transformations, using the compiler's optimization strategy as a starting point. Complementing the compiler strategy with iterative search enables to test a large set of combinations of transformations such as inlining, local padding, loop fusion/fission, loop interchange, loop/register tiling, loop unrolling and prefetching.

Target Benchmarks. We considered five representative SpecFP2000 benchmarks with different behavior, as shown in Figure 6 (`mgrid`, `applu`, `galgel`, `equake`, `apsi`), using the `ref` data sets. We apply optimization only on the most-time consuming sections of these benchmarks. We handpicked these codes based on the study by Parello et al. [26] which suggests which SpecFP2000 benchmarks have the best potential for improvement (on an Alpha 21264 platform, though). Since the role of seed points in iterative search has been previously highlighted [3,34], we also used the latter study as an indication for seed points, i.e., initial points for a space search.

5.2 Results

This section shows that the full power of iterative optimization can be achieved at the cost of profile-directed optimization: one or two runs of each benchmark are sufficient to discover the best optimization options for every phase.

Boosting Search Rate. For each benchmark, Table 2 shows the actual number of evaluated options, which is the number of versions multiplied by the number of instrumented code sections and by the number of phases with significant execution time.

However, the maximum number of potential optimization options (program transformations or compositions of program transformations) that can be evaluated during

Table 2. Maximum number of potential evaluations or iterative search speedup vs. the real number of options evaluated during single execution and the associated overhead

Application	Max. number of potential evaluations	Number of options evaluated	Instrumentation overhead
`mgrid`	699	32	0%
`applu`	430	80	0.01%
`galgel`	32	32	0.01%
`equake`	962	16	13.17%
`apsi`	96	96	0%

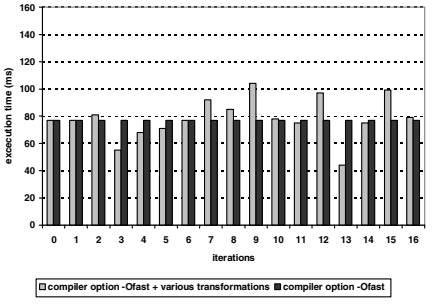


Fig. 8. Execution time variations of the `resid` subroutine of the `mgrid` benchmark over iterations (subroutine calls)

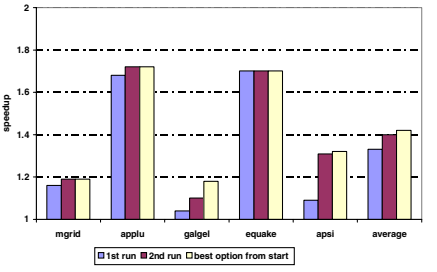


Fig. 9. Speedups of instrumented program over original program after first run, second run or if the best option is selected from the start

one execution of a benchmark can be much higher depending on the application behavior. Thanks to run-time adaptation it is now possible to evaluate 32 to 962 more optimization options than through traditional across-runs iterative optimization. The discrepancy among maximum number of evaluations is explained by the differences in phase behavior of programs and in the instrumentation placement. If the instrumentation is located at a lower loop nest level, it enables a greater number of evaluations but it can also induce excessive overhead and may limit applicable transformations. On the other hand, instrumentation at a higher level enables all spectrum of complex sequences of transformations but can limit the number of potential evaluations during one execution. For example, the number of potential optimization evaluations is small for `galgel` due to chaotic behavior and frequent performance mispredictions, and is high for `equake` due to relatively low level instrumentation.

To quantify the instrumentation overhead, the last column in Table 2 shows the ratio of the execution time of the instrumented program over the original program, assuming all optimization options are replaced with exact copies of the original code section. As a result, this ratio only measures the slowdown due to instrumentation. The overhead is negligible for 4 out of 5 benchmarks, and reaches 13% for `equake`. Note however that `equake` still achieves one of the best speedups at 1.7, as shown in Figure 9.

Self-tuned Speedup. For each selected benchmark we have created a self-tuning program with 16 versions of each most time consuming sections of these benchmarks For each of these versions we applied either combinations of compiler-generated program transformations using our EKOPath compiler API (loop fusion/fission, loop interchange, loop and register tiling, loop unrolling and prefetching with multiple randomly selected parameters) or manual program transformations suggested by Parello et al. [26] for the Alpha platform, or combinations of both. The overall number of evaluations per each benchmark varied from 16 to 96 depending on the number of most-time consuming sections, as shown in table 2.

Figure 8 shows an example of execution time variations for the triple-nested loop in the `resid` subroutine of the `mgrid` benchmark for each option evaluation all within one execution of this program. The baseline performance is shown in a straight gray line

and the best version is found at iteration 13. The final best sequence of transformations for the loop in the subroutine `resid` is loop tiling with tile size 60, loop unrolling with factor 16 and prefetching switched off. The best parameters found for subroutine `psinv` of the same benchmark are 9 for loop tiling and 14 for loop unrolling. Note also that the static algorithm of the EKOPath compiler suggested unrolling factors of 2 for loops of both subroutines. This factor is a kind of tradeoff value across all possible data sets, while the self-tuning code converged toward a different value dynamically. It is important to note that this adjustment occurred at run-time, during a single run, and that the optimization was selected soon enough to improve the remainder of the run.

Figure 9 shows the speedups obtained for all benchmarks after two executions as well as the speedups assuming there is no instrumentation overhead and the best optimization option is used from the start; speedups vary from 1.10 to 1.72. It is interesting to note that, though the manual transformations were designed for a different architecture (Alpha 21264), for 4 out of 5 benchmarks, they were eventually adjusted through transformation parameter tuning to our target architecture, and still perform well. In other terms, beyond performance improvement on a single platform, self-adjusting codes also provide a form of cross-platform portability by selecting the optimization option best suited for the new platform.

6 Related Work

Some of the first techniques to select differently optimized versions of code sections are cloning and multi-versioning [9,14,16]. They use simple version selection mechanisms according to the input run-time function or loop parameters. Such techniques are used to some extent in current compilers but lack flexibility and prediction and cannot cope with various cases where input parameters are too complex or differ while the behavior of the code section remains the same and vice versa.

To improve cloning effectiveness, many studies defer code versioning to the run-time execution. Program hot spots and input/context parameters are detected at run-time, to drive dynamic recompilation. For example, the Dynamo system [5] can optimize streams of native instructions at run-time using hot traces and can be easily implemented inside JIT compilers. Insertion of prefetching instructions or changing prefetching distance dynamically depending on hardware counters informations is presented in [28]. VCODE [18] is a tool to dynamically regenerate machine code, and [4] presents a technique which produces pre-optimized machine-code templates and later dynamically patch those templates with run-time constants. ADAPT [37,36] applies high-level optimizations to program hot spots using dynamic recompilation in a separate process or on a separate workstation and describes a language to write self-tuned applications. Finally, ADORE [10,24] uses a sampling based phase analysis to detect performance bottlenecks and apply simple transformations such as prefetching dynamically.

Recently, software-only solutions have been proposed to effectively detect, classify and predict phase transitions, with very low run-time overhead [5,17]. In particular, [17] decouples this detection from the dynamic code generation or translation process, relying on a separate process sampling hardware performance counters at a fixed

interval. Selection of a good sampling interval is critical, to avoid missing fine-grain phase changes while minimizing overhead [27,29].

In contrast with the above-mentioned projects, our approach is a novel combination of versioning, code instrumentation and software-only phase detection to enable practical iterative evaluation of complex transformations at run-time. We choose static versioning rather than dynamic code generation, allowing low-overhead adaptability to program phases and input contexts. Associating static instrumentation and dynamic detection avoids most pitfalls of either isolated instrumentation-based or sampling-based phase analyses, including sensitivity to calling contexts and sampling interval selection [23]. Finally, we rely on predictive rather than reactive phase detection, although it is not for the reasons advocated in [17]: we do not have to amortize the overhead of run-time code generation, but we need to predict phase changes to improve the evaluation rate for new optimization options.

7 Conclusions and Perspectives

Several practical issues still prevent iterative optimization from being widely used, the time required to search the huge program transformations space being one of the main issues. In this article, we present a method for speeding up search space pruning by a factor of 32 to 962 over a set of benchmarks, by taking advantage of the phase behavior (performance stability) of applications. The method, implemented in the EKOPath compiler, can be readily applied to a large range of applications. The method has other benefits: such self-tuned programs facilitate portability across different architectures and software environments, they can self-adjust at the level of phases and to particular data sets (as opposed to the trade-off proposed by current iterative techniques), they can build a catalog of per-phase appropriate program transformations (code section/performance pairs) across runs, and they can easily combine user-suggested and compiler-suggested transformations thanks to their versioning approach.

Future work will include fast analysis of large complex transformation spaces, improving our phase detection and prediction scheme to capture more complex performance behaviors, and improving the instrumentation placement, especially using self-placement of instrumentation at the most proper loop nest levels, by instrumenting all loop nests levels, then dynamically switching off instrumentation at all levels but one, either using predication or versioning again.

References

1. PAPI: A Portable Interface to Hardware Performance Counters. `http://icl.cs.utk.edu/papi`, 2005.
2. PathScale EKOPath Compilers. `http://www.pathscale.com`, 2005.
3. L. Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 231–239, 2004.
4. J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 149–159, 1996.

5. V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM SIGPLAN Notices*, 2000.
6. J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proc. ICS*, pages 340–347, 1997.
7. F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proc. ACM Workshop on Profile and Feedback Directed Compilation*, 1998. Organized in conjunction with PACT98.
8. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
9. M. Byler, M. Wolfe, J. R. B. Davies, C. Huson, and B. Leasure. Multiple version loops. In *ICPP 1987*, pages 312–318, 2005.
10. H. Chen, J. Lu, W.-C. Hsu, and P.-C. Yew. Continuous adaptive object-code re-optimization framework. In *Ninth Asia-Pacific Computer Systems Architecture Conference (ACSAC 2004)*, pages 241–255, 2004.
11. A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. *ACM Int. Conf on Supercomputing (ICS'05)*, June 2005.
12. K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.
13. K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. of Supercomputing*, 23(1), 2002.
14. K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, pages 96–105, 1992.
15. K. D. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the R^n programming environment. *ACM Transactions on Programming Languages and Systems*, 8:491–523, 1986.
16. P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proc. PLDI*, pages 71–84, 1997.
17. E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *IEEE PACT 2003*, pages 220–231, 2003.
18. D. Engler. Vcode: a portable, very fast dynamic code generation system. In *Proceedings of PLDI*, 1996.
19. G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *Proc. Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.
20. K. Heydeman, F. Bodin, P. Knijnenburg, and L. Morin. Ufc: a global trade-off strategy for loop unrolling for vliw architectures. In *Proc. CPC*, pages 59–70, 2003.
21. S. Hu, M. Valluri, and L. K. John. Effective adaptive computing environment management via dynamic optimization. In *IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005)*, 2005.
22. T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proc. Compilers for Parallel Computers (CPC2000)*, pages 35–44, 2000.
23. J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *International Symposium on High Performance Computer Architecture*, 2005.
24. J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. In *The Journal of Instruction-Level Parallelism*, volume 6, 2004.
25. A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proc. AIMSA*, LNCS 2443, pages 41–50, 2002.

26. D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Toward a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *Proc. Int. Conference on Supercomputing*, 2004.
27. E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. In *ACM SIGMETRICS the International Conference on Measurement and Modeling of Computer Systems*, 2003.
28. R. H. Saavedra and D. Park. Improving the effectiveness of software prefetching with adaptive execution. In *Conference on Parallel Architectures and Compilation Techniques (PACT'96)*, 1996.
29. X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ACM SIGARCH Computer Architecture News*, pages 165–176, 2004.
30. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.
31. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of ASPLOS-X*, 2002.
32. M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005)*. IEEE Computer Society, 2005.
33. M. Stephenson, M. Martin, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proc. PLDI*, pages 77–90, 2003.
34. S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Journal of Instruction-level Parallelism*, 2005.
35. X. Vera, J. Abella, A. González, and J. Llosa. Optimizing program locality through CMEs and GAs. In *Proc. PACT*, pages 68–78, 2003.
36. M. Voss and R. Eigemann. High-level adaptive program optimization with adapt. In *Proceedings of the Symposium on Principles and practices of parallel programming*, 2001.
37. M. Voss and R. Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *Proc. ICPP*, 2000.
38. R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proc. Alliance*, 1998.

Efficient Sampling Startup for Sampled Processor Simulation

Michael Van Biesbrouck[1], Lieven Eeckhout[2], and Brad Calder[1]

[1] CSE Department, University of California, San Diego
[2] ELIS Department, Ghent University, Belgium
{mvanbies, calder}@cs.ucsd.edu, leeckhou@elis.UGent.be

Abstract. Modern architecture research relies heavily on detailed pipeline simulation. Simulating the full execution of an industry standard benchmark can take weeks to months. Statistical sampling and sample techniques like SimPoint that pick small sets of execution samples have been shown to provide accurate results while significantly reducing simulation time. The inefficiencies in sampling are (a) needing the correct memory image to execute the sample, and (b) needing a warm architecture state when simulating the sample.

In this paper we examine efficient Sampling Startup techniques addressing two issues: how to represent the correct memory image during simulation, and how to deal with warmup. Representing the correct memory image ensures the memory values consumed during the sample's simulation are correct. Warmup techniques focus on reducing error due to the architecture state not being fully representative of the complete execution that proceeds the sample to be simulated. This paper presents several Sampling Startup techniques and compares them against previously proposed techniques. The end result is a practical sampled simulation methodology that provides accurate performance estimates of complete benchmark executions in the order of minutes.

1 Introduction

Modern computer architecture research relies heavily on cycle-accurate simulation to help evaluate new architectural features. In order to measure cycle-level events and to examine the effect that hardware optimizations would have on the whole program, architects are forced to execute only a small subset of the program at cycle-level detail and then use that information to approximate the full program behavior. The subset chosen for detailed study has a profound impact on the accuracy of this approximation, and picking these points so that they are as *representative* as possible of the full program is a topic of several research studies [1,2,3,4]. The two bottlenecks in using these sampling techniques are the efficiency of having (a) the correct memory image to execute the sample, and (b) warm architecture state when simulating the sample. We collectively refer to both issues as *Sampling Startup*.

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 47–67, 2005.

1.1 Sample Starting Image

The first issue to deal with is how to accurately provide a sample's starting image. The *Sample Starting Image* (SSI) is the state needed to accurately emulate and simulate the sample's execution to achieve the correct output for that sample[1]. The two traditional approaches for providing the SSI are fast-forwarding and using checkpoints. Fast-forwarding quickly emulates the program's execution from the start of execution or from the last sample to the current sample. The advantage of this approach is that this is trivial for all simulators to implement. The disadvantage is that it serializes the simulation of all of the samples for a program, and it is non-trivial to have a low-overhead fast-forwarding implementation—most fast-forwarding implementations in current simulators are fairly slow.

Checkpointing is the process of storing the program's image right before the sample of execution is to start. This is similar to storing a core dump of the program so that it can be replayed at that point in execution. A checkpoint stores the register contents and the memory state prior to a sample. The advantage of checkpointing is that it allows for efficient parallel simulation. The disadvantage is that if a full checkpoint is taken it can be huge and consume too much disk space and take too long to load.

In this paper we examine two efficient ways of storing the SSI. One is a reduced checkpoint where we only store in the checkpoint the words of memory that are to be accessed in the sample we are going to simulate. The second approach is very similar, but is represented differently. For this approach we store a sequence of executed load values for the complete sample. Both of these approaches take about the same disk space, which is significantly smaller than a full checkpoint. Since they are small they also load instantaneously and are significantly faster than using fast-forwarding and full checkpoints.

1.2 Sample Architecture Warmup

Once we have an efficient approach for dealing with the sample's starting image we also need to reduce as much error in simulation due to the architecture components not being in the same state as if we simulated the full detailed execution from the start of the program up to that simulation point. To address this we examine a variety of previously proposed techniques and compare them to storing the detailed state of the memory hierarchy as a form of architecture checkpoint.

We first examine a technique called "Hit on Cold" which assumes that all architecture components are cold and the first access to it during the sample's simulation is a hit. A second technique we study uses a fixed warmup period before the execution of each sample. Recently, more sophisticated warmup techniques [5,6,7] have focused on finding for each sample how far back in the instruction stream to go to start warming up the architecture structures. We examine

[1] For convenience of exposition, we use 'sample' as a noun to refer to a sampling unit and 'sample' as a verb to refer to collecting a sample unit.

the performance of MRRL [6,7] in this paper. An important advantage of this type of technique is its accuracy. The disadvantage is that it requires architecture component simulation for N million instructions before detailed simulation of the sample, which adds additional overhead to simulation.

The final technique we examine is storing an architecture checkpoint of the major architecture components at the start of the sample. This *Architecture Checkpoint* is used to faithfully recreate the state of the major architecture components, such as caches, TLBs and branch predictors at the start of the sample. It is important that this approach works across different architecture designs for it to be used for architecture design space explorations. To that end, we examine a form of architecture checkpointing that allows us to create the smaller size instances of that architecture component. For example, you would create an architecture checkpoint of the largest cache you would look at in your design space exploration study, and the way we store the architecture checkpoint will allow smaller sizes and associativities to be faithfully recreated.

2 Sampled Simulation Background

Detailed cycle-by-cycle simulation of complete benchmarks is practically impossible due to the huge dynamic instruction counts of today's benchmarks (often several hundred billions of instructions), especially when multiple processor configurations need to be simulated during design space explorations. Sampling is an efficient way for reducing the total simulation time. There exist two ways of sampling, statistical sampling and phase-based sampling.

2.1 Statistical Sampling

Statistical sampling takes a number of execution samples across the whole execution of the program, which are referred to as clusters in [1] because they are groupings of contiguous instructions. These clusters are spread out throughout the execution of the program in an attempt to provide a representative cross-cut of the application being simulated. Conte *et al.* [1] formed multiple simulation points by randomly picking intervals of execution, and then examining how these fit to the overall execution of the program for several architecture metrics (IPC, branch and data cache statistics).

SMARTS [4] provides a version of SimpleScalar [8] using statistical simulation, which uses statistics to tell users how many samples need to be taken in order to reach a certain level of confidence. One consequence of statistical sampling is that tiny samples are gathered over the complete benchmark execution. This means that in the end the complete benchmark needs to be functionally simulated, and for SMARTS, the caches and branch predictors are warmed through the complete benchmark execution. This ultimately impacts the overall simulation time.

2.2 SimPoint

The SimPoint [3] sampling approach picks a small number of samples, that when simulated, accurately create a representation of the complete execution of the program. To do this they break a program's execution into intervals, and for each interval they create a code signature. They then perform clustering on the code signatures grouping intervals with similar code signatures into phases. The notion is that intervals of execution with similar code signatures have similar architecture behavior, and this has been shown to be the case in [3,9,10,11]. Therefore, only one interval from each phase needs to be simulated in order to recreate a complete picture of the program's execution. They then choose a representative from each phase and perform detailed simulation on that interval. Taken together, these samples can represent the complete execution of a program. The set of chosen samples are called *simulation points*, and each simulation point is an interval on the order of millions of instructions. The simulation points were found by examining only a profile of the basic blocks executed for a program.

In this paper we focus on studying the applicability of the Sample Startup techniques presented for SimPoint. In addition, we also provide summary results for applying these Sample Startup techniques to SMARTS.

3 Sampling Startup Related Work

This section discusses prior work on Sample Startup techniques. We discuss checkpointing and fast-forwarding for obtaining a correct SSI, and warmup techniques for obtaining an architecture checkpoint as accurately as possible.

3.1 Starting Sample Image

As stated in the introduction, starting the simulation of a sample is much faster under checkpointing than under fast-forwarding (especially when the sample is located deep in the program's execution trace—fast-forwarding in such a case can take several days). The major disadvantage of checkpoints however is their size; they need to be saved on disk and loaded at simulation time. The checkpoint reduction techniques presented in this paper make checkpointing a much better alternative to fast-forwarding as will be shown in the evaluation section of this paper.

Szwed *et al.* [12] propose to fast-forward between samples through native hardware execution, called direct execution, and to use checkpointing to communicate the application state to the simulator. The simulator then runs the detailed processor simulation of the sample using this checkpoint. When the end of the sample is reached, native hardware execution comes into play again to fast-forward to the next simulation point, *etc.* Many ways to incorporate direct hardware execution into simulators for speeding up the simulation and emulation systems have been proposed, see for example [13,14,15,16].

One requirement for fast-forwarding through direct execution is that the simulation needs to be run on a machine with the same ISA as the program that is to be simulated. One possibility to overcome this limitation for cross-platform simulation would be to employ techniques from dynamic binary translation methods such as just-in-time (JIT) compilation and caching of translated code, as is done in Embra [17], or through compiled instruction-set simulation [18,19]. Adding a dynamic binary compiler to a simulator is a viable solution, but doing this is quite an endeavor, which is why most contemporary out-of-order simulators do not include such functionality. In addition, introducing JITing into a simulator also makes the simulator less portable to host machines with different ISAs. Checkpoints, however, are easily portable.

Related to this is the approach presented by Ringenberg *et al.* [20]. They present intrinsic checkpointing, which takes the SSI image from the previous simulation interval and uses binary modification to bring the image up to state for the current simulation interval. Bringing the image up to state for the current simulation interval is done by comparing the current SSI against the previous SSI, and by providing fix-up checkpointing code for the loads in the simulation interval that see different values in the current SSI versus the previous SSI. The fix-up code for the current SSI then executes stores to put the correct data values in memory. Our approach is easier to implement as it does not require binary modification. In addition, when implementing intrinsic checkpointing one needs to be careful to make sure the fix-up code is not simulated so that it does not affect the cache contents and branch predictor state for warmup.

3.2 Warmup

There has been a lot of work done on warmup techniques, or approximating the hardware state at the beginning of a sample. This work can be divided roughly in three categories: (*i*) simulating additional instructions prior to the sample, (*ii*) estimating the cache miss rate in the sample, and (*iii*) storing the cache content or taking an architecture checkpoint. In the evaluation section of this paper, we evaluate four warmup techniques. These four warmup techniques were chosen in such a way that all three warmup categories are covered in our analysis.

Warmup N Instructions Before Sample - The first set of warmup approaches simulates additional instructions prior to the sample to warmup large hardware structures [1,4,6,7,21,22,23,24,25]. A simple warmup technique is to provide a fixed-length warmup prior to each sample. This means that prior to each sample, caches and branch predictors are warmed by, for example, 1 million of instructions. MRRL [6,7] on the other hand, analyzes memory references and branches to determine where to start warming up caches and branch predictors prior to the current sample. Their goal is to automatically calculate how for back in execution to go before a sample in order to capture the data and branch working set needed for the cache and branch predictor to be simulated. MRRL examines both the instructions between the previous sample and the current sample and the instructions in the sample to determine the correct warmup

period. BLRL [21], which is an improvement upon MRRL, examines only references that are used in the sample to see how far one needs to go back before the sample for accurate warmup.

SMARTS [4] uses continuous warmup of the caches and branch predictors between two samples, *i.e.*, the caches and branch predictor are kept warm by simulating the caches and branch predictor continuously between two samples. This is called functional warming in the SMARTS work. The reason for supporting continuous warming is their small sample sizes of 1000 instructions. Note that continuously warming the cache and branch predictor slows down fast-forwarding.

The warmup approaches from this category that are evaluated in this paper are fixed-length warmup and MRRL.

Estimating the Cache Miss Rate - The second set of techniques does not warm the hardware state prior to the sample but estimates which references in the sample are cold misses due to an incorrect sample warmup [26,27]. These misses are then excluded from the miss rate statistics when simulating the sample. Note that this technique in fact does no warmup, but rather estimates what the cache miss rate would be for a perfectly warmup hardware state. Although these techniques are useful for estimating cache miss rate under sampled simulation, extending these techniques to processor simulation is not straight-forward. The hit-on-cold approach evaluated in this paper is another example of cache miss rate estimation; the benefit of hit-on-cold over the other estimation techniques is its applicability to detailed processor simulation.

Checkpointing the Cache Content - Lauterbach [28] proposes storing the cache tag content at the beginning of each sample. This is done by storing tags for a range of caches as they are obtained from stack simulation. This approach is similar to the Memory Hierarchy State (MHS) approach presented in this paper (see section 5 for more details on MHS). However, there is one significant difference. We compute the cache content for one single large cache and derive the cache content for smaller cache sizes. Although this can be done through stack simulation, it is still significantly slower and more disk space consuming than simulating only one single cache configuration as we do.

The Memory Timestamp Record (MTR) presented by Barr *et al.* [29] is also similar to the MHS proposed here. The MTR allows for the reconstruction of the cache and directory state for multiprocessor simulation by storing data about every cache block. The MTR is largely independent of cache size, organization and coherence protocol. Unlike MHS, its size is proportional to program memory. This prior work did not provide a detailed comparison between their architectural checkpointing approach and other warmup strategies; in this paper, we present a detailed comparison between different warmup strategies.

3.3 SMARTS and TurboSMARTS

Wunderlich *et al.* [4] provide SMARTS, an accurate simulation infrastructure using statistical sampling. SMARTS continuously updates caches and branch

predictors while fast-forwarding between samples of size 1000 instructions. In addition, it also warms up the processor core before taking the sample through the detailed cycle-by-cycle simulation of 2000 to 4000 instructions.

At the same time we completed the research for our paper, TurboSMARTS [30] presented similar techniques that replace functional warming with a checkpointed SSI and checkpointed architectural state similar to what we discuss in our paper. In addition to what was studied in [30], we compare a number of reduced checkpointed SSI techniques, we study the impact of wrong-path load instructions for our techniques, and we examine the applicability of checkpointed sampling startup techniques over different sample sizes.

4 Sample Starting Image

The first issue to deal with to enable efficient sampled simulation is to load a memory image that will be used to execute the sample. The *Sample Starting Image* (SSI) is the program memory state needed to enable the correct functional simulation of the given sample.

4.1 Full Checkpoint

There is one major disadvantage to checkpointing compared to fast-forwarding and direct execution for providing the correct SSI. This is the large checkpoint files that need to be stored on disk. Using many samples could be prohibitively costly in terms of disk space. In addition, the large checkpoint file size also affects total simulation time due to loading the checkpoint file from disk when starting the simulation of a sample and transferring over a network during parallel simulation.

4.2 EIO Files and Checkpointing System Calls

Before presenting our two approaches to reduce the checkpoint file size, we first detail our general framework in which the reduced checkpoint methods are integrated. We assume that the program binary and its input are available through an EIO file during simulation. We use compressed SimpleScalar EIO files; this does not affect the generality of the results presented in this paper however. An EIO file contains a checkpoint of the initial program state after the program has been loaded into memory. Most of the data in this initial program image will never be modified during execution. The rest of the EIO file contains information about every system call, including all input and output parameters and memory updates associated with the calls. This keeps the system calls exactly the same during different simulation runs of the same benchmarks.

In summary, for all of our results, the instructions of the simulated program are loaded from the program image in the EIO file, and the program is not stored in our checkpoints. Our reduced checkpoints focus only on the data stream.

4.3 Touched Memory Image

Our first reduced checkpoint approach is the *Touched Memory Image (TMI)* which only stores the blocks of memory that are to be accessed in the sample that is to be simulated. The TMI is a collection of chunks of memory (touched during the sample) with their corresponding memory addresses. The TMI contains only the chunks of memory that are read during the sample. Note that a TMI is stored on disk for each sample. At simulation time, prior to simulating the given sample, the TMI is loaded from disk and the chunks of memory in the TMI are then written to their corresponding memory addresses. This guarantees a correct SSI when starting the simulation of the sample. A small file size is achieved by using a sparse image representation, so regions of memory that consist of consecutive zeros are not stored in the TMI. In addition, large regions of non-zero sections of memory are combined and stored as one chunk. This saves storage space in terms of memory addresses in the TMI, since only one memory address needs to be stored for a large consecutive data region.

An optimization to the TMI approach, called the *Reduced Touched Memory Image* (*RTMI*), only contains chunks of memory for addresses that are read before they are written. There is no need to store a chunk of memory in the reduced checkpoint in case that chunk of memory is written prior to being read. A TMI, on the other hand, contains chunks of memory for all reads in the sample.

4.4 Load Value Sequence

Our second approach, called the *Load Value Sequence (LVS)*, involves creating a log of load values that are loaded into memory during the execution of the sample. Collecting an LVS can be done with a functional simulator or binary instrumentation tool, which simply collects all data values loaded from memory during program execution (excluding those from instruction memory and speculative memory accesses). When simulating the sample, the load log sequence is read concurrently with the simulation to provide correct data values for non-speculative loads. The result of each load is written to memory so that, potentially, speculative loads accessing that memory location will find the correct value. The LVS is stored in a compressed format to minimize required disk space. Unlike TMI, LVS does not need to store the addresses of load values. However, programs often contain many loads from the same memory addresses and loads with value 0, both of which increase the size of LVS without affecting TMI.

In order to further reduce the size of the LVS, we also propose the *Reduced Load Value Sequence (RLVS)*. For each load from data memory the RLVS contains one bit, indicating whether or not the data needs to be read from the RLVS. If necessary, the bit is followed by the data value, and the data value is written to the simulator's memory image at the load address so that it can be found by subsequent loads; otherwise, the value is read from the memory image and not included in the RLVS. Thus the RLVS does not contain load values when a load is preceded by a load or store for the same address or when the value would be zero (the initial value for memory in the simulator). This yields

a significant additional reduction in checkpoint file sizes. An alternate structure that accomplishes the same task is the first load log presented in [31].

5 Sample Warmup

In this paper we compare five warmup strategies, not performing any warmup, hit on cold, 1M-instructions of detailed execution fixed warmup, MRRL and stored architecture state. The descriptions in this section summarize the warmup techniques in terms of how they are used for uniprocessor architecture simulation.

5.1 No Warmup

The no-warmup strategy assumes an empty cache at the beginning of each sample, *i.e.* assumes no warmup. Obviously, this will result in an overestimation of the number of cache misses, and by consequence an underestimation of overall performance. However, the bias can be small for large sample sizes. This strategy is very simple to implement and incurs no runtime overhead.

5.2 Hit on Cold

The *hit on cold* strategy also assumes an empty cache at the beginning of each sample but assumes that the first use of each cache block in the sample is always a hit. The no warmup strategy, on the other hand, assumes a miss for the first use of a cache block in the sample. Hit on cold works well for programs that have a high hit rate, but it requires modifying the simulator to check a bit on every cache miss. If the bit indicates that the cache block has yet to be used the sample then the address tag is added to the cache but the access is considered to be a hit.

An extension to this technique is to try to determine the overall program's average hit rate or the approximate hit rate for each sample, then use this probability to label the first access to a cache block as a miss or a hit. We did not evaluate this approach for this paper.

5.3 Memory Reference Reuse Latency

The *Memory Reference Reuse Latency (MRRL)* [6,7] approach proposed by Haskins and Skadron builds on the notion of memory reference reuse latency. The memory reference reuse latency is defined as the number of dynamic instructions between two consecutive memory references to the same memory location. To compute the warmup starting point for a given sample, MRRL first computes the reuse latency distribution over all the instructions from the end of the previous sample until the end of the current sample. This distribution gives an indication about the temporal locality behavior of the references.

MRRL subsequently determines w_N which corresponds to the $N\%$ percentile over the reuse latency distribution. This is the point at which $N\%$ of memory

references, between the end of the last sample to the end of the current sample, will be made to addresses last accessed within w_N instructions. Warming then gets started w_N instructions prior to the beginning of the sample. The larger $N\%$, the larger w_N, and thus the larger the warmup. In this paper we use $N\% = 99.9\%$ as proposed in [6].

Sampled simulation under MRRL then proceeds as follows. The first step is to fast-forward to or load the checkpoint at the starting point of the warmup simulation phase. From that point on until the starting point of the sample, functional simulation is performed in conjunction with cache and branch predictor warmup, *i.e.* all memory references warm the caches and all branch addresses warm the branch predictors. When the sample is reached, detailed processor simulation is started for obtaining performance results. The cost of the MRRL approach is the w_N instructions that need to be simulated under warmup.

5.4 Memory Hierarchy State

The fourth warmup strategy is the *Memory Hierarchy State (MHS)* approach, which stores cache state so that caches do not need to be warmed at the start of simulation. The MHS is collected through cache simulation, *i.e.* functional simulation of the memory hierarchy. Design-space exploration may require many different cache configurations to be simulated. Note that the MHS needs to be collected only once for each block size and replacement policy, but can be reused extensively during design space exploration with smaller-sized memory hierarchies. Our technique is similar to trace-based construction of caches, except that storing information in a cache-like structure decreases both storage space and time to create the cache required for simulation. In addition to storing cache tags, we store status information for each cache line so that dirty cache lines are correctly marked.

Depending on the cache hierarchies to be simulated, constructing hierarchies of *target caches* for simulation from a single *source cache* created by functional simulation can be complicated, so we explain the techniques used and the necessary properties of the source cache. If a target cache i has s_i sets with w_i ways, then every cache line in it would be contained in a source cache with $s' = c_i s_i$ sets and $w' \geq w_i$ ways, where c_i is a positive integer. We now describe how to initialize the content of each set in the target cache from the source cache. To initialize the contents of the first set in the target cache we need to look at the cache blocks in c_i sets of the source cache that map to the first set in the target cache. This gives us $c_i w'$ blocks to choose from. For a cache with LRU replacement we need to store a sequence number for the most recent use of each cache block. We select the most recently used w_i cache blocks, as indicated by the data in our source cache, to put in the target set. The next c_i sets of the source cache initialize the second set of the target cache, and so forth. In general, $s' = \texttt{LCM}_i(s_i)$ and $w' = \texttt{max}_i(w_i)$ ensure that the large cache contains enough information to initialize all of the simulated cache configurations. In the common case where s_i is always a power of two, the least common multiple (LCM) is just the largest such value.

Inclusive cache hierarchies can be initialized easily as just described, but exclusive cache hierarchies need something more. For example, assume that the L2 cache is 4-way associative and has the same number of cache sets as a 2-way associative L1 cache. Then the 6 most recently accessed blocks mapped to a single cache set will be stored in the cache hierarchy, 2 in the L1 cache and 4 in the L2 cache. Thus the source cache must be at least 6-way associative. If, instead, the L2 cache has twice as many sets as the L1 cache then the cache hierarchy will contain 10 blocks that are mapped to the same L1 cache set, but at most 6 of these will be mapped to either of the L2 cache sets associated with the L1 cache set. The source cache still only needs to be 6-way associative.

We handle exclusive cache hierarchies by creating the smallest (presumably L1) target cache first and locking the blocks in the smaller cache out of the larger (L2, L3, *etc.*) caches. Then the sets in the larger cache can be initialized. Also, the associativity of the source cache used to create our MHS must be at least the sum of the associativities of the caches within a target cache hierarchy.

Unified target caches can be handled by collecting source cache data as if the target caches were not unified. For example, if there are target IL1, DL1 and UL2 caches then data can be collected using the same source caches as if there were IL2 and DL2 caches with the same configuration parameters as the UL2 cache. Merging the contents of two caches into a unified target cache is straight-forward. The source caches must have a number of cache sets equal to the LCM of all the possible numbers of cache sets (both instruction and data) and an associativity at least as large as that of any of the caches. Alternately, if all of the target cache hierarchy configurations are unified in the same way then a single source cache with the properties just described can collect all of the necessary data.

Comparing MHS versus MRRL, we can say that they are equally architecture-independent. MHS traces store all addresses needed to create the largest and most associative cache size of interest. Similarly, MRRL goes back in execution history far enough to also capture the working set for the largest cache of interest. The techniques have different tradeoffs, however: MHS requires additional memory space compared to MRRL, and MRRL just needs to store where to start the warming whereas MHS stores a source cache. In terms of simulation speed, MHS substantially outperforms MRRL as MHS does not need to simulate instructions to warm the cache as done in MRRL—loading the MHS trace is done very quickly. As simulated cache sizes increase, MHS disk space requirements increase and MRRL warming times increase.

The techniques discussed in this paper can also be extended to warmup for TLBs and branch predictors. For 1M-instruction simulation points, we only consider sample architecture warmup for caches. We found that the branch predictor did not have a significant effect until we used very small 1000-instruction intervals with SMARTS. When simulating the tiny SMARTS simulation intervals we checkpointed the state of branch predictor prior to each sample. While we can modify TLB and cache checkpoints to work with smaller TLBs and caches, we cannot yet do this in general for branch predictors. For experiments requiring

branch predictor checkpoints we simulate all of the branch predictors to be used in the design space exploration concurrently and store their state in the sample checkpoints.

6 Evaluation

We now evaluate Sample Startup for sampled simulation. After discussing our methodology, we then present a detailed error analysis of the warmup and reduced checkpointing techniques. We subsequently evaluate the applicability of the reduced checkpointing and warmup techniques for both targeted sampling as done in SimPoint and statistical sampling as done in SMARTS.

6.1 Methodology

We use the MRRL-modified SimpleScalar simulator [6], which supports taking multiple samples interleaved with fast-forwarding and functional warming. Minor modifications were made to support (reduced) checkpoints. We simulated SPEC 2000 benchmarks compiled for the Alpha ISA and we used reference inputs for all of these. The binaries we used in this study and how they were compiled can be found at `http://www.simplescalar.com/`. The processor model assumed in our experiments is summarized in Table 1.

6.2 Detailed Error Analysis

We first provide a detailed error analysis of our warmup techniques as well as the reduced checkpointing techniques. For the error analysis, we consider 50 1M-instruction sampling units randomly chosen from the entire benchmark execution. We also experimented with a larger number of sampling units, however, the results were quite similar.

The *average CPI error* is the average over all samples of the relative difference between the CPI through sampled simulation with full warmup, versus the CPI through sampled simulation with the warmup and reduced checkpoint techniques proposed in this paper. Our second metric is the *95% confidence interval* for average CPI error. Techniques that are both accurate and precise will have low average CPI error and small confidence interval widths.

Table 1. Processor simulation model

I Cache	8k 2-way set-assoc., 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-assoc., 32 byte blocks, 2 cycle latency
L2 Cache	1Meg 4-way set-assoc., 32 byte blocks, 20 cycle latency
Memory	150 cycle round trip access
Branch Pred	hybrid
O-O-O Issue	up to 8 inst. per cycle, 128 entry re-order buffer
Func Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV

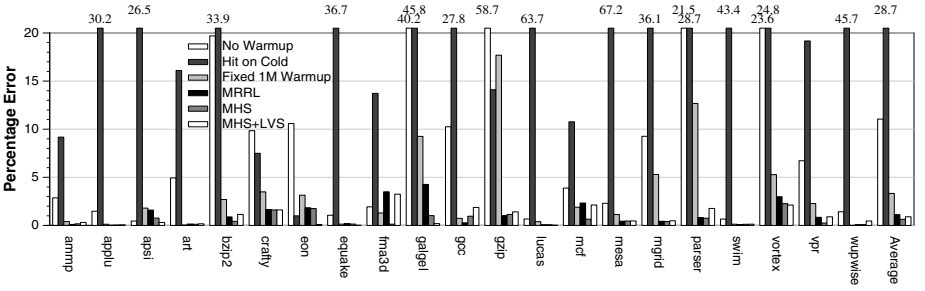


Fig. 1. Average CPI error: average CPI sample error as a percentage of CPI

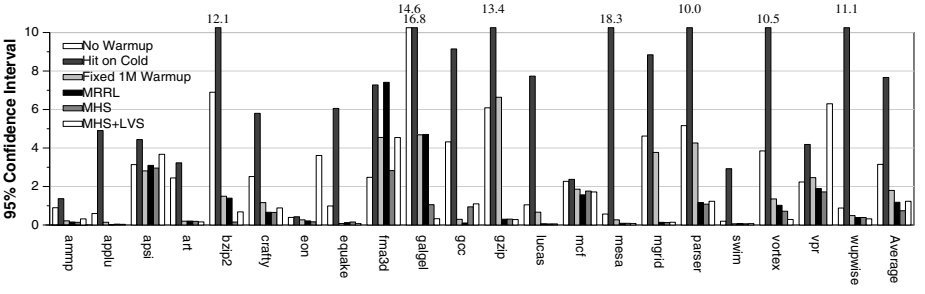


Fig. 2. The 95% confidence interval as a percentage of CPI

Figures 1 and 2 show the average CPI error and the 95% confidence interval, respectively. In both cases they are expressed as a percentage of the correct CPI. The various bars in these graphs show full SSI checkpointing along with a number of architectural warmup strategies (no warmup, hit on cold, fixed 1M warmup, MRRL and MHS), as well as a reduced SSI checkpointing technique, namely LVS, in conjunction with MHS. We only present data for the LVS reduced SSI for readability reasons; we obtained similar results for the other reduced SSI techniques. In terms of error due to warmup, we find that the no-warmup, hit-on-cold and fixed 1M warmup strategies perform poorly. MRRL and MHS on the other hand, are shown to perform equally well. The average error is less than a few percent across the benchmarks.

In terms of error due to the starting image, we see the error added due to the reduced SSI checkpointing is very small. Comparing the MHS bar (MHS with full checkpointing) versus the MHS+LVS bar, we observe that the error added is very small, typically less than 1%. The reason for this additional error is that under reduced SSI checkpointing, load instructions along mispredicted paths might potentially fetch wrong data from memory since the reduced checkpointing techniques only consider on-path memory references. In order to quantify this we refer to Figures 3 and 4. Figure 3 shows the percentage of wrong-path load instructions being issued relative to the total number of issued loads; this figure also shows the percentage of issued wrong-path loads that fetched incorrect data

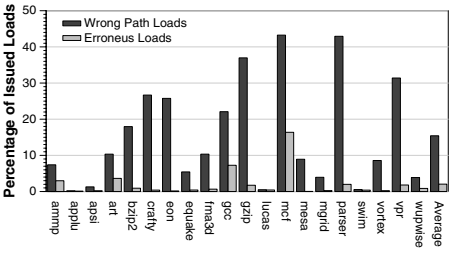


Fig. 3. Analysis of wrong-path loads while using LVS

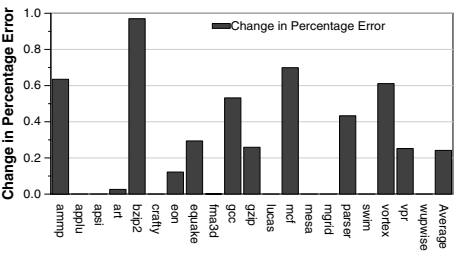


Fig. 4. Change in percentage error due to LVS

(compared to a fully checkpointed simulation) relative to the total number of issued loads. This graph shows that the fraction of wrong-path loads that are fetching uncheckpointed data is very small, 2.05% on average. Figure 4 then quantifies the difference in percentage CPI error due to these wrong-path loads fetching uncheckpointed data. We compare the CPI under full checkpoint versus the CPI under reduced checkpoints. The difference between the error rates is very small, under 1% of the CPI.

6.3 Targeted Sampling Using SimPoint

We now study the applicability of the reduced warmup and checkpointing techniques for two practical sampling methodologies, namely targeted sampling using SimPoint, and statistical sampling using SMARTS. For these results we used SimPoint with an interval size of 1 million with Max K set to 400. Figure 5 shows the number of 1M-instruction simulation points per benchmark. This is also the number of checkpoints per benchmark since there is one checkpoint needed per simulation point. The number of checkpoints per benchmark varies from 15 (`crafty`) up to 369 (`art`). In this paper, we focus on small, 1M-instruction intervals because SimPoint is most accurate when many (at least 50 to 100) small (1M instructions or less) intervals are accurately simulated. However, we found that the reduction in disk space is an important savings even for 10M and 100M

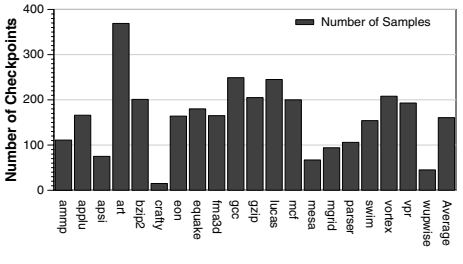


Fig. 5. Number of Simulation Point samples used

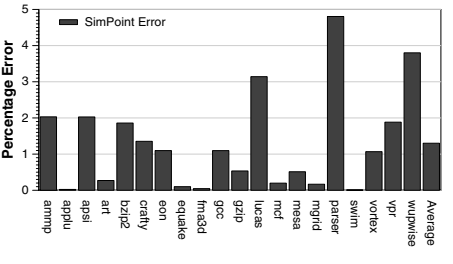


Fig. 6. Accuracy of SimPoint assuming perfect sampling

interval sizes when using a reduced load value trace. Figure 6 shows the accuracy of SimPoint while assuming perfect sampling startup. The average error is 1.3%; the maximum error is 4.8% (`parser`).

Storage Requirements - Figures 7 and 8 show the average and total sizes of the files (in MB) that need to be stored on disk per benchmark for various Sample Startup approaches: the Full Checkpoint, the Load Value Sequence (LVS), the Reduced Load Value Sequence (RLVS), the Touched Memory Image (TMI) and the Memory Hierarchy State (MHS). Clearly, the file sizes for Full Checkpoint are huge. The average file size per checkpoint is 49.3MB (see Figure 7). The average total file size per benchmark is 7.4GB (see Figure 8). Storing all full checkpoints for a complete benchmark can take up to 28.8GB (`lucas`). The maximum average storage requirements per checkpoint can be large as well, for example 163.6MB for `wupwise`. Loading and transferring over a network such large checkpoints can be costly in terms of simulation time as well.

The SSI techniques, namely LVS, RLVS, TMI and RTMI, result in a checkpoint reduction of more than two orders of magnitude, see Figures 7 and 8. The results for RTMI are similar to those for TMI, so they are not shown in the Figure. Since TMI contains at most one value per address and no zeros, size improvements can only come from situations where the first access to an address

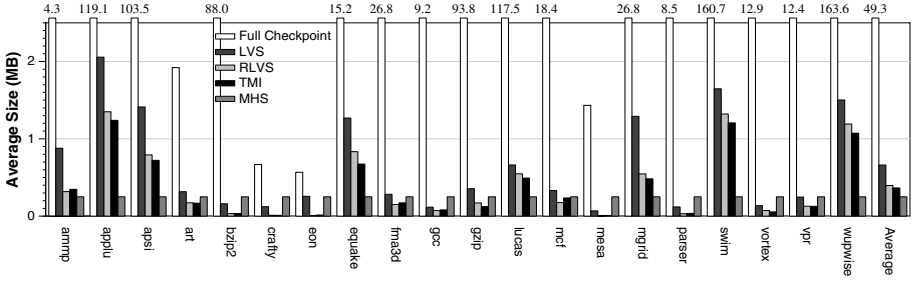


Fig. 7. Average storage requirements per sample for full checkpointing, reduced checkpointing (LVS, RLVS and TMI) and sample warmup through MHS

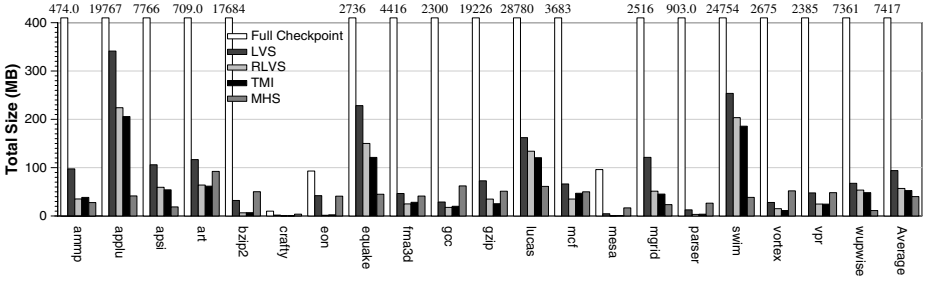


Fig. 8. Total storage requirements per benchmark for full checkpointing, reduced checkpointing (LVS, RLVS and TMI) and sample warmup through MHS

is a write and there is a later read from that address. This is fairly rare for the benchmarks examined, so the improvements are small.

The average total checkpoint file sizes per benchmark for LVS, RLVS and TMI are 93.9MB, 57MB and 52.6MB, respectively; the maximum total file sizes for are 341MB, 224MB and 206MB, respectively, for `applu`. These huge checkpoint file reductions compared to full checkpoints make checkpointing feasible in terms of storage cost for sampled simulation. Also, the typical single checkpoint size is significantly reduced to 661KB, 396KB and 365KB for LVS, RLVS and TMI, respectively. This makes loading the checkpoints highly efficient.

Memory Hierarchy State (MHS) was the only warmup approach discussed that requires additional storage. Figures 7 and 8 quantify the additional storage needed for MHS. The total average storage needed per benchmark is 40MB. The average storage needed for MHS per checkpoint is 256kB (8 bytes per cache block). Note that this is additional storage that is needed on top of the storage needed for the checkpoints. However, it can be loaded efficiently due to its small size.

Error Analysis - Figure 9 evaluates the CPI error rates for various Sample Startup techniques as compared to the SimPoint method's estimate using perfect warmup — this excludes any error introduced by SimPoint. This graph compares four sample warmup approaches: no warmup, hit-on-cold, fixed 1M warmup, MRRL and MHS. The no warmup and hit-on-cold strategies result in high error rates, 16% and 24% on average. For many benchmarks one is dramatically better than the other, suggesting that an algorithm that intelligently chooses one or the other might do significantly better. The fixed 1M warmup achieves better accuracy with an average error of 4%; however the maximum error can be fairly large, see for example for `parser` (17%). The error rates obtained from MRRL and MHS are significantly better. The average error for both approaches is 1%. As such, we conclude that in terms of accuracy, MRRL and MHS are equally accurate when used in conjunction with SimPoint.

The error results discussed so far assumed full checkpoints. Considering a reduced checkpoint technique, namely LVS, in conjunction with MHS increases the error rates only slightly, from 1% to 1.2%. This is due to the fact that LVS does not include load values for loads being executed along mispredicted paths.

Total Simulation Time - Figure 10 shows the total simulation time (in minutes) for the various Sample Startup techniques when simulating all simulation points on a single machine. This includes fast-forwarding, loading (reduced) checkpoints, loading the Memory Hierarchy State and warming structures by functional warming or detailed execution, if appropriate.

The SSI techniques considered here are fast-forwarding, checkpointing, and reduced checkpointing using the LVS—we obtained similar simulation time results for the other reduced checkpoint techniques RLVS, TMI and RTMI. These three SSI techniques are considered in combination with the two most accurate sample warmup techniques, namely MRRL and MHS. These results show that MRRL in combination with fast-forwarding and full checkpointing are equally

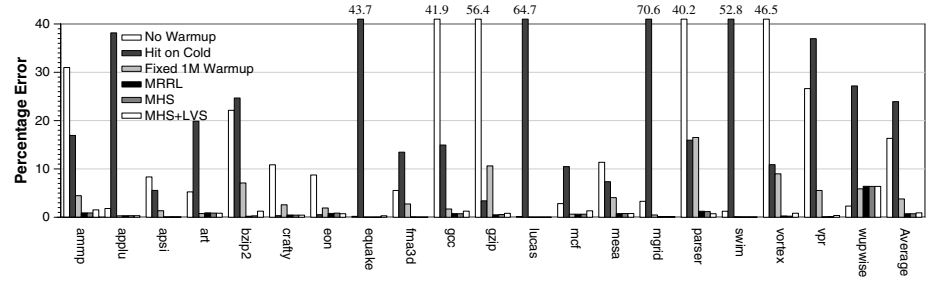


Fig. 9. Percentage error in estimating overall CPI as compared to CPI estimated by SimPoint with no sampling error

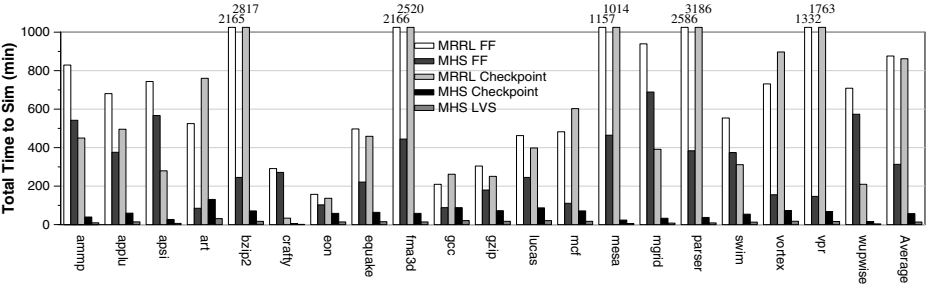


Fig. 10. Total time to simulate all samples including fast-forwarding, loading checkpoints, warming and doing detailed simulation

slow. The average total simulation time is more than 14 hours per benchmark. If we combine MHS with fast-forwarding, the average total simulation time per benchmark cuts down to 5 hours. This savings over MRRL is achieved by replacing warming with fast-forwarding and loading the MHS. Combining MHS with full checkpointing cuts down the total simulation time even further to slightly less than one hour. Combining the reduced checkpoint LVS approach with MHS reduces the average total simulation time per benchmark to 13 minutes. We obtained similar simulation time results for the other reduced checkpoint techniques.

As such, we conclude that the Sample Startup techniques proposed in this paper achieve full detailed per-benchmark performance estimates with the same accuracy as MRRL. This is achieved in the order of minutes per benchmark which is a 63X simulation time speedup compared to MRRL in conjunction with fast-forwarding and checkpointing.

6.4 Using MHS and LVS with SMARTS

The SMARTS infrastructure [4] accurately estimates CPI by taking large numbers of very small samples and using optimized functional warming while

fast-forwarding between samples. Typical parameters use approximately 10000 samples, each of which is 1000 instructions long and preceded by 2000 instructions of detailed processor warmup. Only 30M instructions are executed in detail, so simulation time is dominated by the cost of functional warming for tens or hundreds of billions of instructions.

We improved SMARTS' performance by replacing functional warming with our MHS and LVS techniques. Due to the very small sample length there was insufficient time for the TLB and branch predictor to warm before the end of detailed simulation warmup. Therefore, for SMARTS we enhanced MHS to include the contents of the TLBs and branch predictor. TLB structures can be treated just like caches when considering various TLB sizes, but branch predictors need to be generated for every desired branch predictor configuration. With these changes we were able to achieve sampling errors comparable to the error rates presented in section 6.2 for the 1M-instruction samples. In addition, the estimated CPI confidence intervals are similar to those obtained through SMARTS.

Storing the entire memory image in checkpoints for 10000 samples is infeasible, so we used LVS. Due to the small number of loads in 3000 instructions, a compressed LVS only required a single 4 kB disk block per sample. The total disk space per benchmark for the LVS checkpoint is 40 MB. Disk usage however is dominated by MHS, with total storage requirements of approximately 730 MB for each benchmark. By comparison, our SimPoint experiments used under 100 MB on average for full LVS, 50 MB for RLVS and 40 MB for MHS. In terms of disk space, SimPoint thus performs better than SMARTS. On average, the total simulation time per benchmark for SMARTS with LVS and MHS is 130 seconds on average. About two-thirds of this time is due to decompressing the MHS information. Our results assumed 10000 samples, but more samples may be needed in order to reach a desired level of confidence, which would require more simulation time.

In contrast to the fact that the amount of disk space required is approximately 8 times larger with SMARTS, SMARTS is faster than SimPoint: 130 seconds for SMARTS versus 13 minutes for SimPoint. The reason for this is the larger of number of simulated instructions for SimPoint than for SMARTS.

Concurrently with our work, the creators of SMARTS have released TurboSMARTS [30], which takes a similar approach to the one that we have outlined here. Their documentation for the new version recommends estimating the number of samples that should be taken when collecting their version of MHS and TMI data. The number of samples is dependent upon the program's variability, so for floating-point benchmarks this can greatly reduce the number of samples, but in other cases more samples will be required. As a result, the average disk usage is 290 MB per benchmark, but varies from 7 MB (`swim`) to 1021 MB (`vpr`). This is still over twice as large than the disk space required for SimPoint using 1 million interval sizes.

7 Summary

Today's computer architecture research relies heavily on detailed cycle-by-cycle simulation. Since simulating the complete execution of an industry standard benchmark can take weeks to months, several researchers have proposed sampling techniques to speed up this simulation process. Although sampling yields substantial simulation time speedups, there are two remaining bottlenecks in these sampling techniques, namely efficiently providing the sample starting image and sample architecture warmup.

This paper proposed reduced checkpointing to obtain the sample starting image efficiently. This is done by only storing the words of memory that are to be accessed in the sample that is to be simulated, or by storing a sequence of load values as they are loaded from memory in the sample. These reduced checkpoints result in two orders of magnitude less storage than full checkpointing and faster simulation than both fast-forwarding and full checkpointing. We show that our reduced checkpointing techniques are applicable on various sampled simulation methodologies as we evaluate them for SimPoint, random sampling and SMARTS.

This paper also compared four techniques for providing an accurate hardware state at the beginning of each sample. We conclude that architecture checkpointing and MRRL perform equally well in terms of accuracy. However, our architecture checkpointing implementation based on the Memory Hierarchy State is substantially faster than MRRL. The end result for sampled simulation is that we obtain highly accurate per-benchmark performance estimates (only a few percent CPI prediction error) in the order of minutes, whereas previously proposed techniques required multiple hours.

Acknowledgments

We would like to thank the anonymous reviewers for providing helpful comments on this paper. This work was funded in part by NSF grant No. CCR-0311710, NSF grant No. ACR-0342522, UC MICRO grant No. 03-010, and a grant from Intel and Microsoft. Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (F.W.O. Vlaanderen); and he is a member of the HiPEAC network of excellence.

References

1. Conte, T.M., Hirsch, M.A., Menezes, K.N.: Reducing state loss for effective trace sampling of superscalar processors. In: ICCD'96. (1996)
2. Lafage, T., Seznec, A.: Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In: WWC-3. (2000)
3. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: ASPLOS-X. (2002)

4. Wunderlich, R.E., Wenisch, T.F., Falsafi, B., Hoe, J.C.: SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In: ISCA-30. (2003)
5. Eeckhout, L., Eyerman, S., Callens, B., De Bosschere, K.: Accurately warmed-up trace samples for the evaluation of cache memories. In: HPC'03. (2003) 267–274
6. Haskins, J., Skadron, K.: Memory reference reuse latency: Accelerated sampled microarchitecture simulation. In: ISPASS'03. (2003)
7. Haskins, J., Skadron, K.: Accelerated warmup for sampled microarchitecture simulation. ACM Transactions on Architecture and Code Optimization (TACO) **2** (2005) 78–108
8. Burger, D.C., Austin, T.M.: The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison (1997)
9. Lau, J., Sampson, J., Perelman, E., Hamerly, G., Calder, B.: The strong correlation between code signatures and performance. In: ISPASS. (2005)
10. Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., Karunanidhi, A.: Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In: MICRO-37. (2004)
11. Yi, J.J., Kodakara, S.V., Sendag, R., Lilja, D.J., Hawkins, D.M.: Characterizing and comparing prevailing simulation techniques. In: HPCA-11. (2005)
12. Szwed, P.K., Marques, D., Buels, R.M., McKee, S.A., Schulz, M.: SimSnap: Fast-forwarding via native execution and application-level checkpointing. In: INTERACT-8. (2004)
13. Durbhakula, M., Pai, V.S., Adve, S.: Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors. In: HPCA-5. (1999)
14. Fujimoto, R.M., Campbell, W.B.: Direct execution models of processor behavior and performance. In: Proceedings of the 1987 Winter Simulation Conference. (1987) 751–758
15. Mukherjee, S.S., Reinhardt, S.K., B. Falsafi, M.L., Huss-Lederman, S., Hill, M.D., Larus, J.R., Wood, D.A.: Wisconsin wind tunnel II: A fast and portable parallel architecture simulator. In: PAID'97. (1997)
16. Schnarr, E., Larus, J.R.: Fast out-of-order processor simulation using memoization. In: ASPLOS-VIII. (1998)
17. Witchel, E., Rosenblum, M.: Embra: Fast and flexible machine simulation. In: SIGMETRICS'96. (1996) 68–79
18. Nohl, A., Braun, G., Schliebusch, O., Leupers, R., Meyr, H., Hoffmann, A.: A universal technique for fast and flexible instruction-set architecture simulation. In: DAC-41. (2002)
19. Reshadi, M., Mishra, P., Dutt, N.: Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In: DAC-40. (2003)
20. Ringenberg, J., Pelosi, C., Oehmke, D., Mudge, T.: Intrinsic checkpointing: A methodology for decreasing simulation time through binary modification. In: ISPASS'05. (2005)
21. Eeckhout, L., Luo, Y., De Bosschere, K., John, L.K.: Blrl: Accurate and efficient warmup for sampled processor simulation. The Computer Journal **48** (2005) 451–459
22. Conte, T.M., Hirsch, M.A., Hwu, W.W.: Combining trace sampling with single pass methods for efficient cache simulation. IEEE Transactions on Computers **47** (1998) 714–720
23. Kessler, R.E., Hill, M.D., Wood, D.A.: A comparison of trace-sampling techniques for multi-megabyte caches. IEEE Transactions on Computers **43** (1994) 664–675

24. Luo, Y., John, L.K., Eeckhout, L.: Self-monitored adaptive cache warm-up for microprocessor simulation. In: SBAC-PAD'04. (2004) 10–17
25. Nguyen, A.T., Bose, P., Ekanadham, K., Nanda, A., Michael, M.: Accuracy and speed-up of parallel trace-driven architectural simulation. In: IPPS'97. (1997) 39–44
26. Laha, S., Patel, J.H., Iyer, R.K.: Accurate low-cost methods for performance evaluation of cache memory systems. IEEE Transactions on Computers **37** (1988) 1325–1336
27. Wood, D.A., Hill, M.D., Kessler, R.E.: A model for estimating trace-sample miss ratios. In: SIGMETRICS'91. (1991) 79–89
28. Lauterbach, G.: Accelerating architectural simulation by parallel execution of trace samples. In: Hawaii International Conference on System Sciences. (1994)
29. Barr, K.C., Pan, H., Zhang, M., Asanovic, K.: Accelerating multiprocessor simulation with a memory timestamp record. In: ISPASS'05. (2005)
30. Wenisch, T.F., Wunderlich, R.E., Falsafi, B., Hoe, J.C.: TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. In: SIGMETRICS. (2005)
31. Narayanasamy, S., Pokam, G., Calder, B.: Bugnet: Continuously recording program execution for deterministic replay debugging. In: ISCA. (2005)

Enhancing Network Processor Simulation Speed with Statistical Input Sampling

Jia Yu[1], Jun Yang[1], Shaojie Chen[2], Yan Luo[1], and Laxmi Bhuyan[1]

[1] Department of Computer Science and Engineering
[2] Department of Statistics,
University of California, Riverside
{jiayu, junyang, yluo, bhuyan}@cs.ucr.edu,
schen009@ucr.edu

Abstract. While cycle-accurate simulation tools have been widely used in modeling high-performance processors, such an approach can be hindered by the increasing complexity of the simulation, especially in modeling chip multi-processors with multi-threading such as the network processors (NP). We have observed that for NP cycle level simulation, several days of simulation time covers only about one second of the real-world network traffic. Existing approaches to accelerating simulation are through either code analysis or execution sampling. Unfortunately, they are not applicable in speeding up NP simulations due to the small code size and the iterative nature of NP applications. We propose to sample the *traffic input* to the NP so that a long packet trace is represented by a much shorter one with simulation error bounded within $\pm 3\%$ and 95% confidence. Our method resulted one order of magnitude improvement in the NP simulation speed.

1 Introduction

Network processors have emerged as a solution to programmable routers with high processing capability. To better understand the effectiveness of the NP architecture, there have been a number of analytical models to quantify both the architecture and the energy features of an NP [2,3]. Compared with those analytical models, a cycle-accurate NP simulator can faithfully emulate micro-architectural behaviors, which is more appropriate for optimizing designs in order to achieve better processing capability and energy efficiency. However, cycle-accurate simulation of NPs is extremely time consuming. For example, one second of IXP 2400 hardware execution (600M cycles) corresponds to 10 days of simulation time in the Intel SDK 4.0 environment [1] on a Intel Xeon 3GHz PC with 512 MB memory in our experiment[1]. Similar results are observed even in a much lightweight simulator that we developed previously [8]. Apparently, the slow simulation speed limits its advantages for architectural optimizations and design space exploration over a large time scale.

There have been a number of techniques proposed to accelerate architectural simulation speed. The prevailing methodologies are: 1) truncated execution — terminating

[1] This is obtained from running the "OC12_pos_gbeth_2401" included in Intel SDK 4.0 with one Ethernet and one POS port under uniform arrival rate.

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 68–83, 2005.

execution at a specified point; 2) using reduced input sets — using a smaller input instead of a large input for shorter execution; 3) SimPoint [4] — a tool that selects and simulates representative segments of a program (termed simulation points) via the analysis of the execution frequencies of basic blocks. The intuition behind it is that the overall behavior of the entire execution can be characterized by certain segments of the program executing an input. Machine learning techniques are used to divide execution intervals into *clusters*, and one or more points from each cluster for simulation. The final results are weighted summarized from each cluster result. One challenging issue how to choose the number of clusters K. In [16], Perelman *et al.* proposed a *Variance SimPoint* algorithm which used statistical analysis to guide the selection of K such that a K-clustering will meet a given level of confidence and probabilistic error bound. 4) sampling — the simulation intervals are sampled for execution from the entire simulation duration. Conte *et al.* [15] first applied statistical sampling in processor simulations in which the confidence and the probabilistic error bounds are used to control the accuracy. Another more rigorous statistical sampling method, SMARTS [5], assembles many detailed small simulation points with fast-forwarding (i.e. functional simulations) between them. This approach achieved low error bounds of $\pm 3\%$ on average with a high confidence of 99.7%. A recent study [7] showed that SMARTS and SimPoint are both very accurate, while truncated execution and using reduced input sets have poor accuracy.

However, methodologies in SimPoint and SMARTS cannot be directly applied in NP simulations. The major reasons are: 1) **different benchmark characteristics**. Unlike SPEC2K benchmarks that have large and complex execution paths, the NP benchmarks periodically execute the same programs for processing multiple packets. Also, a single run of the NP program (processing a single packet) is very short, varying from hundreds to tens of thousands cycles only. With such short code paths, there is no need for applying SimPoint to find representative execution points, because most of the instructions are representative. 2) **different architectures and programming models**. For SPEC2K, both SimPoint and SMARTS are applicable for single threaded simulation. However, the NP architecture is typically multi-cored with multi-threading. Each core may be parallel or pipelined with its neighboring cores. Thus, the overall NP performance not only depends on the individual execution path in each core, but also the interactions among the cores and threads. 3) **different simulation focuses**. The standard input sets for SPEC2K are typically used without adaption. Thus results depend more on the program itself. In NP simulation, the metrics depend on not only the application code, but also the incoming traffic workload, e.g. low vs. high traffic volume, Ethernet vs. WWW traffic. Different traffic inputs will cause varying execution paths, and consequently tremendously different architecture states. Therefore, the input traffic analysis is more important for NP performance estimation than the code analysis.

We propose to conduct the input data analysis for accelerating NP simulations, complementary to the existing simulation techniques. We evaluate the effects of different input parameters to the NP architecture states, and observe that redundant simulation do exist. We show theoretically as well as confirm experimentally that sampling network inputs can significantly reduce the input sizes, while still bounding the error. In order to estimate the variation of the collected results, we used a similar approach as in the *Variance SimPoint* [16] algorithm with the difference in that we perform clustering on

the input. Also, there is no need to consider warmups or fast-forwarding issues during the sample simulation due to the specialty of NP applications. This is because skipping packets in the input traces has nearly no affect to the future packet processing due to the low locality in the packet payloads. With our methodology, we can achieve one order of magnitude speed up for the NP micro-architecture simulation.

In the next section, we discuss the characteristics of NP simulation inputs and the correlation between them and performance metrics. In section 3, we study the variations of NP simulation properties. In section 4, the stratified input sampling approach, as well as simple random sampling and systematic sampling are presented. Section 5 presents the results of proposed sampling methods. Section 6 concludes the paper.

2 Correlation Between Input and NP Performance Metrics

2.1 NP Programming Model and Benchmark Applications

The programming model for a typical NP follows the receive-process-transmit paradigm [1]. The receive task collects the incoming packets from the network interfaces and reassembles sub-packets when necessary. The process task executes certain application functions to the packets such as packet filtering and address translation. The transmit task segments the packets and instructs the hardware to send them to the correct network port. The three tasks are mapped to different PEs with queues latching packet handles between them. There are four benchmarks ported to the original NePSim [8]. We compiled and ported three more cryptographic applications [9] for our experiments, and the descriptions are listed in table 1. More details about the benchmarks can be found in [8,9].

Table 1. Description of the NP benchmarks

Benchmarks	Description
Ipfwdr	IPv4 Ethernet and IP header validation and trie-based routing-table lookup.
Nat	Network address translation, retrieve a replacement address and port for packets.
Url	Route packets on the bases of their contained URL request.
Md4	Cryptographic algorithm that produce a 128-bit fingerprint, or digital signature.
AES	Cryptographic algorithm incorporated into 802.11i
Blowfish	Cryptographic algorithm incorporated into Norton Utilities
RC4	Cryptographic algorithm incorporated into SSL/TSL, 802.1x

2.2 Characterization of Simulation Inputs

The major traffic parameters that affect the execution paths of packet processing are *packet size*, *arrival rate*, *packet source/destination addresses*, *protocol types*. In the following, we will discuss how much these parameters affect the simulation results.

– *Packet size*. Larger packet sizes usually cause longer packet processing time. For header processing applications, only headers will be decapsulated and processed, so larger packet size does not add much work to the processing stage. However,

the receive and transmit stages are lengthened since longer time is spent moving packets between the bus interface and the packet buffer. For payload processing applications, larger packet sizes translate to more work in payload processing, and thus they cause longer processing time.

To show the relationship between the packet size and packet processing time, we run the seven benchmarks in NePSim with increasing packet sizes from 64 to 1088 bytes (i.e. 1 to 17 *mpackets*, and *mpacket* is the unit packet processing size in IXP family NPs), as shown in Figure 1 and 2. All other parameters are fixed. The arrival rate is 1000 packets/second/port with 16 device ports in total. We observe that average packet latencies increase with the packet sizes in all the seven benchmarks, and the cryptographic benchmarks *aes, blowfish, rc4* show linear increasing trend. This is because they spend most of the time in processing the packet payload stored in SDRAM. Thus the number of SDRAM accesses increases linearly with packet sizes for cryptographic benchmarks.

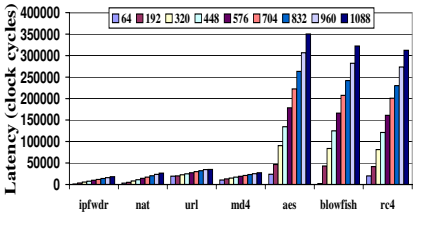


Fig. 1. Average packet latency of different packet sizes

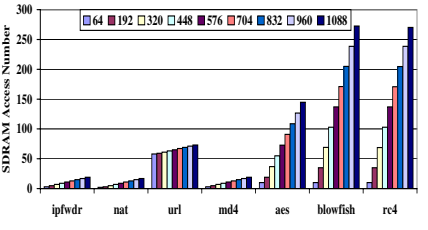


Fig. 2. Average number of SDRAM accesses per packet

– *Packet arrival rate*. This parameter affects the NP architecture states in several ways. First, the threads execute different parts of the code under different arrival rates. At low traffic workload, the threads might be busy polling the *packet descriptor queues* to see if a new packet is ready. This is the case with functional pipelining[1] where the PEs are interleaved with *packet descriptor queues*. Under high traffic volumes, the threads work mostly on packet processing. Since the code segments for packet polling and processing show different computation requirements, the simulation results for low/high traffic arrival rates are different. Second, under high traffic volumes, the average memory access latency can also be increased because of higher demands on the memory and the longer queuing time when the memory is saturated. As a result, the total packet processing time becomes longer.

Figure 3 shows the relationship between the packet arrival rate and packet latency. Similarly, we only vary the arrival rate from 0.1Mbps, 1Mbps, 10Mbps to 100Mbps per port with 16 ports in total. The packet size is set fixed as 128 bytes, and the remaining parameters are unchanged. As we can see, the packet latencies do not change much until the benchmarks approach their maximum throughput. For *ipfwdr* and *nat*, their maximum throughput are approximately 1.3Gbps. Therefore, we observe longer packet latency when the input traffic is 1.6Gbps. For the

payload applications, we observe longer packet latency at 160Mbps arrival rate, because their maximum throughput are around 150Mbps. Figure 4 and 5 report the SDRAM and SRAM average access latencies at different packet arrival rates. The memory latency here includes the memory queuing time, and it increases significantly when the inputs approach the applications' maximum throughput. The memory here more or less becomes the performance bottleneck.

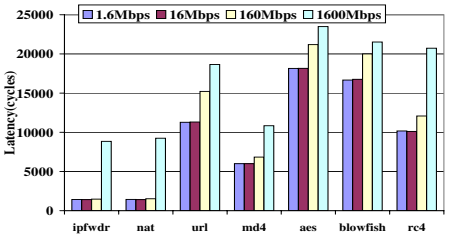


Fig. 3. Average packet latencies (128B/pkt)

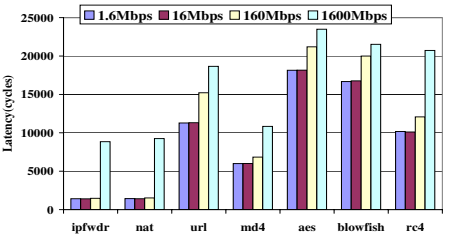


Fig. 4. Average SDRAM access latencies (128B/pkt)

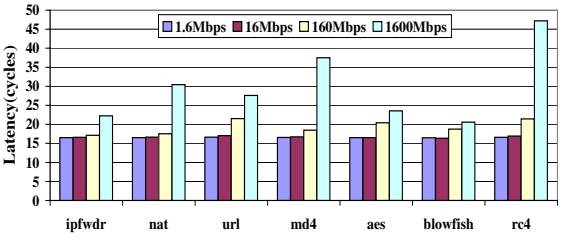


Fig. 5. Average SRAM access latencies (128B/pkt)

– *Packet source/destination addresses and protocol types*. Different values in packet headers trigger different execution paths. Packet source/destination addresses are important information for applications that need to perform routing table lookups such as the IP forwarding benchmark. However, through experiments, we found the variation in destination addresses only cause at most 2% variation in packet latencies for *ipfwdr*. This is because the percentage of the code for address look up is relatively small. Therefore, the variance in the destination addresses does not affect the NP performance significantly.

The NP can be programmed to process packets of different protocol types. However, this does not necessarily cause significant differences in NP architecture states when packets of one protocol type dominates. Take the *ipfwdr* as an example, the code length and complexity for different protocols are usually similar. The execution paths of processing different IP packets only differ by a small amount in the packet header decapsulation part, while the IP address lookup and exception checking/handling parts are the same. Therefore, in our input sampling technique, we do not consider the IP packet distribution from different network protocols.

2.3 Correlation Between Inputs and NP Performance Metrics

In this section, we demonstrate that the NP architecture states are correlated with the network traffic inputs. Only if they are correlated, can we say a particular traffic pattern corresponds to a certain architecture state (e.g. throughput, packet latency). Thus by identifying representative subsets of traffic, we can skip the simulation of those packets that produce similar architecture states.

To quantitatively measure the correlation, we divide the simulation time to n windows with window size as 0.01 second (will be explained in section 4). We then calculate the *correlation coefficient* r [13], for data pairs: (arrival rate, packet loss), (arrival rate, packet latency), (arrival rate, PE idle time) and (arrival rate, IPC).

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}}, i = 1, ..., n \quad (1)$$

$r > 0$ means the data pairs are *positively correlated*, and $r < 0$ means that data pairs are *negatively correlated*. A value of $|r| > 0.8$ means that the data pairs have strong linear relation. A value of $|r| < 0.3$ means that the linear relation is relatively weak. We plot the values of r for *nat* given a set of arrival rates (from 0 to 1800Mbps with 128 bytes per packet and Poisson distribution for the packet arrival rate in Figure 6. Other benchmarks show similar trend as Figure 6.

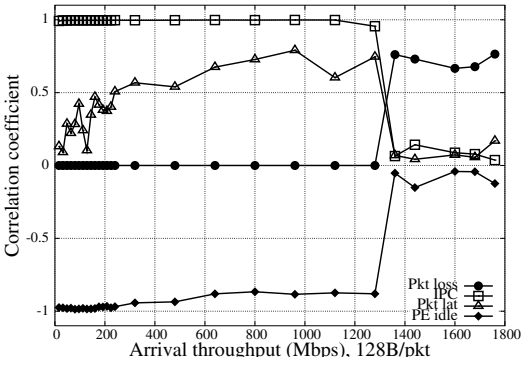


Fig. 6. Correlation of arrival rate and NP metrics (Poisson arrival pattern, 128B/pkt)

We can see from Figure 6 that there are three sections where different metrics show different degrees of correlation with the packet arrival rate. Under low traffic volumes (<300Mbps), there is no packet loss and thus this metric has no correlation with the packet arrival rate. The packet latency has weak correlation with the arrival rate, which means a little variation of arrival rate at light traffic volume does not cause much difference in the average packet latency. The other two metrics, IPC and the PE idle time are both strongly correlated with the packet arrival rate. IPC increases and the PE idle time decreases almost linearly with the increasing packet arrival rates.

With medium traffic volume (300Mbps<packet arrival rate<1300Mbps), the correlation stays about the same for all the metrics except for the packet latency. The *r* value for data pairs (arrival rate, packet latency) approaches to or exceeds 0.8. This means the packet latency has relatively strong correlation with the packet arrival rate at medium traffic volume which is the typical case for an NP.

With high traffic volume (packet arrival rate>1300Mbps), all the metrics we measure exhibit different correlations with the packet arrival rate. First of all, the benchmark *nat* reaches its maximum throughput at 1300Mbps so some of the incoming packets will be dropped. At this time, the input traffic already saturates the NP processing capability, so all metrics (IPC, packet latency, PE idle time) no longer fluctuate with the input except for the packet loss. This metric starts to become positively linear to the packet arrival rate.

From this study, we know that the performance results (IPC, packet latency and PE idle time) under medium to high traffic volume are linearly correlated to the inputs. Thus sampling inputs at medium to high traffic volume can effectively help calculate the corresponding architecture states.

3 Variation of NP Simulation Properties

In this section, we will examine the variation of architecture states, and its relationship with the variation of traffic input. The study of the architecture states is important because their variation is used to estimate the required sample sizes to achieve a certain confidence level. A large variation corresponds to a large sample size.

According to the sampling theory [13], the estimate of a property *x* such as the IPC, average packet latency etc. can be represented as:

$$\bar{X} \pm z \times \frac{\sigma_X}{\sqrt{n}} \tag{2}$$

where $\bar{X}$ is the *mean* value of property *x* in simulation, *z* is percentile of the standard normal distribution (*z*=2.0 for 95% and *z*=3.0 for 99.7% confidence), *n* is the *sample size*, and σ_X is the *standard deviation* of property *x* which depends on both simulation configurations and the input workload. To measure the sampling variation of the estimate relative to the mean of the property *x* being estimated, it is a standard practice to use the *coefficient of variation* V_x ($V_x = \frac{\sigma_X}{\bar{X}}$). Large V_x means the sampled values vary significantly around the mean value $\bar{X}$, thus a large sample size is needed. Small V_x means the sampled values are quite homogeneous, thus a small size is enough. Formula 2 can also be written as:

$$\bar{X} \pm z \cdot \frac{V_x}{\sqrt{n}} \cdot \bar{X} = \bar{X} \cdot (1 \pm z \cdot \frac{V_x}{\sqrt{n}}) \tag{3}$$

To bound the error rate $z \cdot \frac{V_x}{\sqrt{n}}$ at *confidence interval* $[-\varepsilon \cdot \bar{X}, \ +\varepsilon \cdot \bar{X}]$ and ε is a confidence percentile chosen by users, e.g. 3%, 5%, the sample size *n* should be changed proportionally to V_x^2, $n \geq (\frac{z \cdot V_x}{\varepsilon})^2$. A large V_x requires a large sample size *n* to meet a certain confidence. The coefficient of variation V_x (*x* is a performance metric) is rarely

available unless we run the full simulation. Often, an estimation of V_x, $\hat{V}_x$ is used. For example, in SMARTS, $\hat{V}_x$ is estimated from a large initial sample set, as long as the IPC in later periods do not vary significantly. In NP simulation, a large initial sample set cannot estimate the true V_x accurately because the traffic volume after the initial period may change significantly. Correspondingly, the value of V_x will be different. Next, we analyze the possibilities and the challenges of estimating the V_x.

We define $V_{performance}$ as $MAX((V_{x_1}, V_{x_2}, \cdots, V_{x_m})$, where x_1, x_2 ... x_m represent our target simulation metrics (e.g. packet loss ratio, packet latency etc.). Hence, the V_xs are subsumed in the $V_{performance}$. Recall from what was discussed in section 2, the variation of an architecture metric is correlated with the variation of the input traffic, or the workload of the NP. Let $V_{workload}$ be $MAX(V_{arrival\_rate}, V_{packet\_size}...)$, which can be easily obtained by parsing the input traffic. $V_{workload}$ is correlated with $V_{performance}$ and serve as an estimate to V_x.

If $V_{workload} \geq V_{performance}$, we can conservatively use $V_{workload}$ to replace $V_{performance}$. Hence, the calculated sample size $\hat{n} = (\frac{z \cdot V_{workload}}{\varepsilon})^2 \geq (\frac{z \cdot V_{performance}}{\varepsilon})^2 = n$. In other words, taking $\hat{n}$ elements can bound the error rate at a certain confidence level.

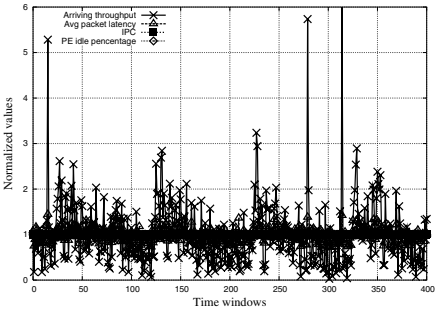


Fig. 7. The variation of the packet arrival rate is significantly larger than variation of NP performance metrics for *url* under relatively low traffic volume (40Mbps). The Y axis is normalized to the mean value.

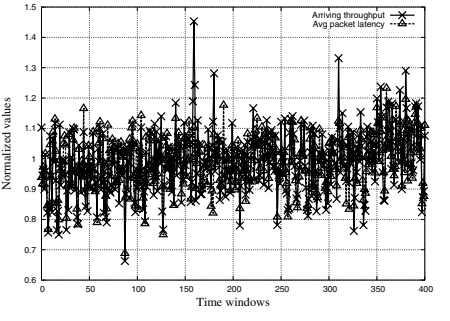


Fig. 8. The variation of the packet arrival rate is close to the variation of packet latency for *rc4* under low traffic volume (40Mbps). The Y axis is normalized to the mean value.

To see if $V_{workload} \geq V_{performance}$ is generally true, we run simulations in NePSim with different input traces. All the input traces are extracted from real world traces from NLANR[12]. Each trace contains 50 second network packets (roughly 500K to 2500K packets). For all our targeting NP metrics, The relationship between $V_{workload}$ and $V_{performance}$ can be categorized into three groups:

– *Group 1:* $V_{workload} > V_{performance}$. Some performance metrics such as IPC and PE activities (execution, stall, idle, abort) vary less significantly than the traffic volume. This phenomenon happens because of the special NP architecture and programming model. When there is a traffic spike, the IPC and PE activities do not change dramatically because the internal buffering scheme in NP's receive-process-transmit model smooths out them effectively when the traffic volume is not overly high.

The packet latency in relatively low traffic volume shows less variation than the traffic input also. Figure 7 plots such an example. In this figure, the traffic volume varies between 0 to 6x of the mean value. However, the corresponding average packet latency only varies within a small window: [0.5x, 1.5x].

– *Group 2:* $V_{workload} \approx V_{performance}$. For cryptographic benchmarks, the variation of the packet latency is very close to $V_{workload}$. Figure 8 shows the normalized variation of packet latency with respect to the packet arrival rate for *rc4*. We can see that the packet latency changes proportionally to its arrival rate.

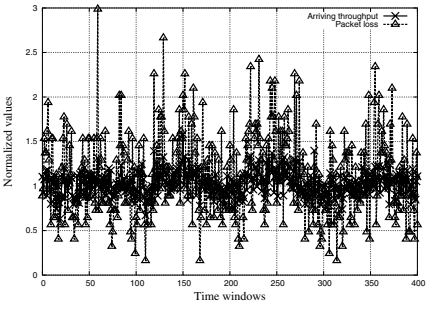


Fig. 9. The variation of the packet arrival rate is less than the variation of packet loss for *ipfwdr* under high traffic volume (1000Mbps). The Y axis is normalized to the mean value.

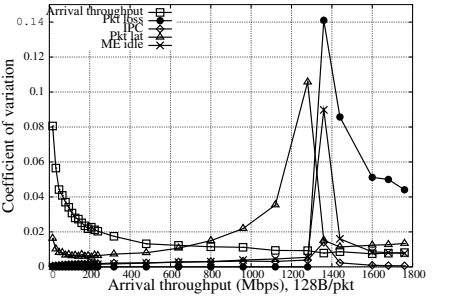


Fig. 10. The variation of all the metrics for *nat* with inputs that follow the Poisson distribution

– Group 3: $V_{workload} < V_{performance}$. At a high traffic volume, we observe that the $V_{performance}$ might be larger than $V_{workload}$, especially for packet loss and packet latency. Figure 9 shows that the packet loss metric has higher variation than input arrival rate, when the arrival rate approaches 1000Mbps for *ipfwdr*. This is because if the NP is close to saturation, the queuing systems in NP lengthen the service latency significantly (as shown in Figure 3 to 5), which magnifying the variations of internal performance metrics.

If all the simulations belong to group 1 and 2, we can safely use $V_{workload}$ to calculate a sample size. However, we did find that some simulation periods belong to group 3. In Figure 10, we plot the $V_{workload}$ and V_x of different metrics for *nat* with packet arrival rate varying from 0 to 1800Mbps. We observe that under low and extremely high traffic volumes, the coefficient of variance of our targeting metrics tend to be small. These periods belong to group 1, and a small sample size is enough for a good simulation. However, with medium to high traffic volume, the variation of the NP performance can be larger than the variation of the input. Using $V_{workload}$ to replace $V_{performance}$ might underestimate the sample size and introduce inaccuracy. Therefore, we cannot completely replace $V_{performance}$ with $V_{workload}$, and we still need to estimate the $V_{performance}$ from an initial simulation. The difference between ours and SMARTS is that we extract the initial simulation points according to the distribution of the workload. We will illustrate our method in the next section.

4 Statistical Input Sampling Technique

Developing an input sampling mechanism for NP simulation requires the evaluation of the trade-off between the sample size and the simulation accuracy. Our goal is to find a sample size that accurately describes the entire population of the packets. As we discussed earlier, $V_{performance}$ cannot be known unless a full simulation is conducted. Here we propose a two-phase sampling process to solve this problem. The idea of this process is to get *an estimation of* $V_{performance}$, $\hat{V}_{performance}$ through an initial simulation. The input for the initial simulation is sampled from original input trace, according to $V_{workload}$. Recall that $V_{workload}$ is easily estimated by parsing the input trace. If the initial sample size is large enough, $\hat{V}_{performance}$ will be quite close to $V_{performance}$.Thus we can use $\hat{V}_{performance}$ to estimate a more accurate sample size. The process of two-phase sampling mechanism is illustrated in Figure 11.

In the first phase we conduct a pre-survey through an initial simulation. We use $V_{workload}$ to estimate the initial sample size n_1. We then feed the initially sampled packets to the simulator and calculate $\hat{V}_{performance}$ as an estimation of $V_{performance}$. If $\hat{V}_{performance} > V_{workload}$, we use $\hat{V}_{performance}$ to estimate the accurate sample size n_2. We sample the additional $n_2 - n_1$ elements from the input trace and simulate them. If $\hat{V}_{performance} \leq V_{workload}$, we know enough elements have been simulated, and the additional sampling is not necessary. The post processing stage will multiplex the sampled simulation results to get the final results. In the following, we will describe the input sampling using three methods: *simple random, systematic random* and *stratified random samplings*, and discuss their trade-off.

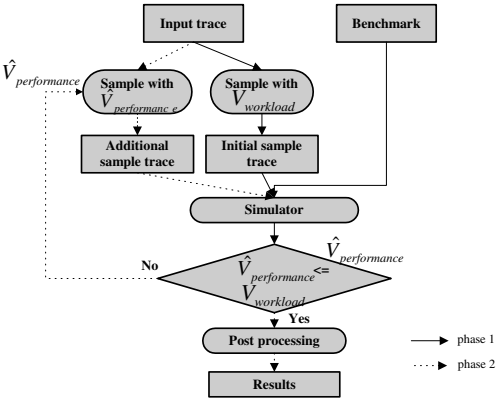


Fig. 11. The two-phase input sampling methodology for NP simulation

4.1 Simple Random and Systematic Random Samplings

Suppose the full input trace has N-second of packets. Each sample contains U-second of traffic packets. If there are n samples, we get the total sample size as $n \cdot U$.

The *simple random sampling* uniformly selects n periods of traffic packets from the full input trace at random. The *systematic random sampling* selects the sample periods

at a fixed sampling interval k such that $n = N/k$. As explained in section 3, to bound confidence interval $\pm\varepsilon \cdot \bar{X}$, the sample size should satisfy:

$$n \geq (\frac{z \cdot V_x}{\varepsilon})^2 \tag{4}$$

4.2 Stratified Random Sampling

Simple random sampling and systematic sampling, each involves taking a sampling from a population as a whole, neither requires identification of subdomains or subgroups before the sample is taken [10]. In other words, they do not differentiate between the high traffic volume and low traffic volume. Additionally, they do not exploit the frequent similar behaviors in network traffic traces, resulting many redundant sample collecting. This observation motivates us to first partition the traffic inputs into groups or strata (low, medium, high traffic etc.), and sample separately within each stratum. The resulting sampling design is the *stratified random sampling*. The purpose of stratification is to minimize the intra-stratum variance while maximizing the inter-stratum variance. Thus a small sample size within each stratum can meet the desired confidence. In addition, the sample sizes taken from strata can be non-uniform, depending on the variations in the strata.

The process of stratified trace sampling is illustrated in Figure 12. We first cluster the traffic trace into K strata based on the packet arrival rate and the packet size using the K-means algorithm [11]. The K-means algorithm is one of the fastest clustering algorithms. It clusters the elements according to their distances to the cluster centroid, and iteratively change the cluster centroid until element membership cease to change. The ratio between the *intra-cluster* and *inter-cluster coefficient of variations*, denoted by β_{CV} [14], is a useful guide to determining the quality of the clustering process. The smaller the value of β_{CV}, the better the clustering. From statistical experience, we choose the stratum size K from 3 to 15 so that the sample sizes in different strata are not too small. Within a particular stratum h, we select a sample of n_h elements from the N_h elements in the stratum, and each element is measured with respect to some variable x (i.e. throughput, packet size).

Stratified Trace Sampling

1. Divide the traffic trace to N periods, measure the throughput and average packet size of each period.
2. Cluster the N periods using K-means algorithm according to the throughput and average packet size.
   a. Select the number of strata $K(3 \leq K \leq 15)$ that produces strata with less than 90% of minimum β_{CV} and sample size.
   b. Calculate required sample size in each stratum, based on the strata sizes strata variations and desired confidence interval.
3. Sample the traffic trace in each stratum and calculate weight of each sampled period, which is proportional to the total number of elements in the stratum.

Fig. 12. Process of stratified trace sampling

To calculate n_h, we apply the *optimal sample allocation* method[6] [10]. The intuition behind this method is that the optimal number of sample elements to be taken from a given stratum is proportional to N_h, the total number of elements in the stratum, and to σ_{hx}, the standard deviation of x among all elements in the stratum. To ensure that the central limit theorem [10] holds, n_h should not be too small. In our implementation, we constrain the minimum sample size for each stratum to be 30.

The stratified trace sampling should generate *weights* for each sampled element, because sampled elements selected from larger stratum should be of greater importance. The weights will be provided to NP simulator, which will scale the simulation result in the post processing stage. We define *weight* of a sampled element as $\frac{N_h}{n_h}$, and the estimated performance metric x can be represented as

$$\sum_{i=1}^{N} X_i \approx \sum_{h=1}^{K} \sum_{i=1}^{n_h} \frac{X_i}{n_h} \cdot N_h = \sum_{h=1}^{K} \sum_{i=1}^{n_h} X_i \cdot weight_i \tag{5}$$

4.3 The Sample Unit Size Selection

The choice of sample unit size U will affect the variation among the periods, and consequently the total sample size. Therefore, we should use U that produces smaller sample sizes. In addition, the U value should be larger than per packet processing time. Next, we show an example of how to determine the value of U.

Figure 13 shows the traffic volume of a 12-hour traffic trace. We divide it into 24 half-hour traces and use different sample unit size $U = 0.001sec$, $0.01sec$, $0.1sec$, $1sec$, $10sec$ to sample each trace. As we can see from Figure 15, the *coefficients of variation* approximately reduce from 1.8 to 0.25 as U increases. This phenomenon is intuitive because the network spikes or dips are smoothed out over longer periods. When $U \geq 0.1sec$, we observe the required sample sizes increase significantly. This is because U scales much faster than the reduction of n (in proportion to ${V_x}^2$). In this example, $U = 0.1sec$ is suitable for sampling because it achieves small sample size and it is

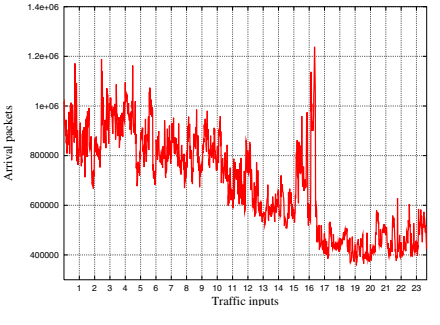


Fig. 13. Traffic volume of Leipzig trace 2002-11-12 from NLANR, the packets arrived in every 30 seconds

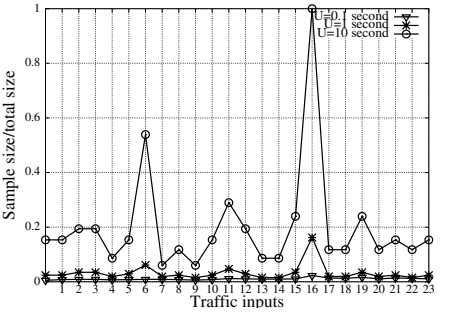


Fig. 14. Required sample sizes to achieve $\pm$ 3% error with 95% confidence in systematic sampling using 0.1sec, 1sec 10sec sampling unit size

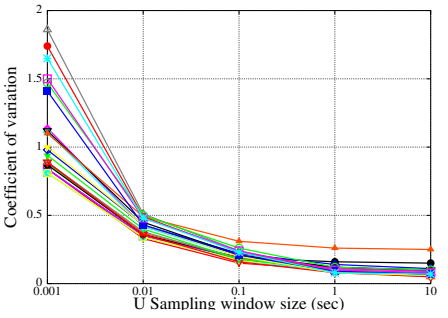


Fig. 15. The coefficients of variation of the traces with various sample unit sizes

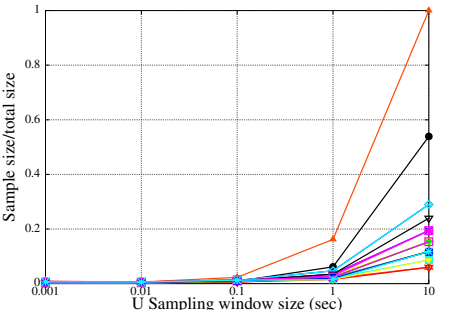


Fig. 16. Required sample sizes to achieve ±3% error rate with 95% confidence

significantly longer than per packet processing time. For comparison purpose, we plot the required sample size for the 24 traces in Figure 14. We can see that the required sample size is correlated with the traffic shape in Figure 13. The network spikes appear in trace 16 and 6 on the x-axis, so do the sample sizes.

4.4 Comparison of the Three Sampling Methods

Simple random and systematic random sampling are comparatively easy to implement, and they are useful to sample traces that have low simulation variations. Inputs with low or extremely high traffic volumes belong to this category. Stratified sampling is effective to reduce the sample size for traces with medium to high traffic volume. In simple random sampling case, usually we need a larger sample size to achieve a specified confidence level because we know little information about the whole population. When the population is highly non-homogeneous, the stratified sampling usually performs better than the systematic random sampling and simple random sampling [10].

5 Experiment

In this section, we present the experiment results of input sampling. Since the three sampling methods can all produce accurate results with large sample sizes, we will compare in two ways: 1) sample sizes for achieving same accuracy 2) for the same sample size, how much accuracy attainable for the three sampling methods. We run the full simulation for the original traces, and compare the results with simulations of sampled traces to get the error rates. We used six inputs extracted from three real world NLANR [12] traces *Leipzig-I, Abilene-I and Tera-I*. *Leipzig-I* is an OC-3 trace collected from an edge access link. *Abilene-I* is an OC48 trace collected at an Indianapolis core router. *Tera-I* is an OC192 10GigE ethernet trace. Each of the six inputs is 50-second long, and costs 2 days for full simulation in NePSim. Two of the input traces are low or medium traffic volume (less than 300Mbps), the other four are high traffic load where packet loss occurs (higher than 300Mbps). They have different $V_{workload}$, varying from 0.1 to 0.8. For sample size estimation, we bound the error rate at 3% within 95% confidence. We

use 0.01 second as sample unit size, because the tested trace is only 50 second long and $U = 0.01$ can effectively reduce sample size.

We expect the simulation time in terms of core cycles will be proportional to input trace length. The speedup attainable is dependent on the sample size, and thus is affected by the architectural variation and total trace length. An extreme case is that the incoming packets are uniformly distributed. In that case, one element is enough to estimate the overall performance, so the speedup would be very large. However, real world traces are rarely uniform, so the speedup attainable is lower.

Figure 17 plots the speedups achieved for simulating the six traces over the three sampling techniques for a common error bound and confidence interval. The systematic or simple random samplings can only achieve 1.8x speedup. While using stratified sampling, the input traces can averagely be reduced by 11 times. We do not observe higher speedups, because the original traces are short and we constrain the sample size in each stratum to be at least 30 (due to central limit theory).

Next, we show the results when the same sample size is used across different sampling techniques. Since stratified sampling requires the smallest sample size among

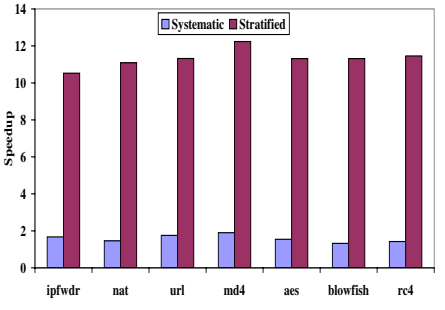


Fig. 17. Speedup achieved using two input sampling methods within ±3% error rate with 95% confidence. *Simple random sampling* has similar speedup as systematic sampling.

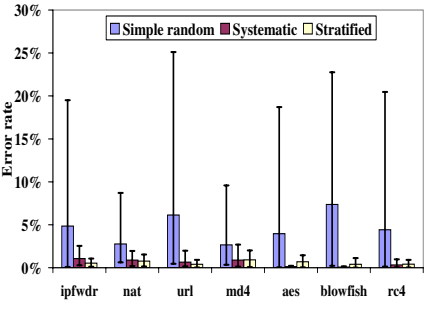


Fig. 18. Error rate of throughput using a same sample size

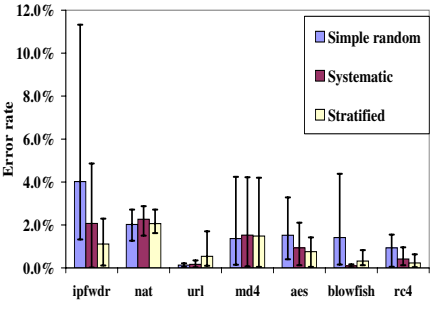


Fig. 19. Error rate of simulated average packet latency using a same sample size

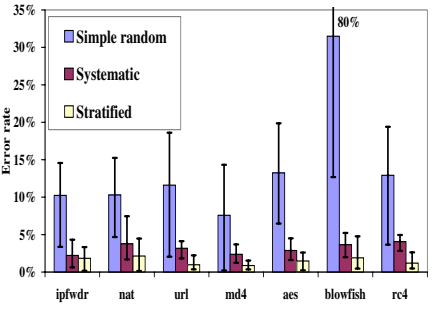


Fig. 20. Error rate of simulated packet loss ratio using a same sample size

the three sampling methods, we use this smallest sample size for the other sampling methods and compare the accuracy. Figure 18 shows the error rate of traffic throughput processed by an NP. Both systematic and stratified sampling can achieve less than 3% error. The simple random sampling produces higher error rate, e.g. 7% on average for *url*. This shows the given sample size is not enough for simple random sampling. Figure 19 shows the error rate of average packet latency. We can see that the three sampling methods are bounded within 3% error for most benchmarks. This is because packet latency has comparatively smaller variance than throughput, thus the given sample size is enough. For *md4*, we observe 4% error rate for simulating one trace. We find the population of this trace is rather homogeneous according to its variance. In this case, the three sampling methods do not differ greatly. In addition, the error rate of *md4* is not bounded by 3%. It is due to the bias of the estimation of $V_{performance}$ in the pre-survey process for this particular trace.

Figure 20 shows the error rate of simulated packet loss ratio. The simple random sampling has error rates larger than 5%. In worst case, the error rate even achieves 80% for a highly non-homogeneous input. For the systematic sampling the average error rates for *nat, url, blowfish, rc4* are larger than 3%. It shows that systematic sampling requires larger sample size to achieve 3% error rate than stratified sampling. Regarding the remaining performance metrics, such as PE idle percentage and IPC, all three sampling methods can achieve less than 3% error rate, because these metrics have smaller variation than packet loss and throughput. In this set of experiments, we observe that stratified sampling performs better than the other two sampling methods for traces that have large variations.

6 Conclusion

We developed a statistical input sampling methodology for accelerating the NP simulations.We experimented several sampling techniques and conclude that the stratified sampling results in the smallest sample size and best accuracy. The speedups we obtained is dependent on the length of the input trace. The longer the trace, the higher the speedup. The outcome of this study can be used for fast estimating NP activities and network performance metrics for non-saturated and non-uniform network traffic. It can also be applied for measurements exploiting dynamic optimizations for better performance and energy efficiency.

References

1. Intel Corporation. “Intel IXP2XXX Product Line of Network Processors.“ http://www.intel.com/design/network/products/ npfamily/ixp2xxx.htm
2. S. Lakshmanamurthy, K.Y. Liu and Y. Pun. “Network Processor Performance Analysis Methodology,” *Intel Technology Journal*, vol. 06, 2002.
3. M.A. Franklin and T. Wolf, “Power Considerations in Network Processor Design,” *Workshop on Network Processors in conjunction with HPCA-9*, 2003.
4. T. Sherwood, E. Perelman, G. Hamerly and B. Calder. “Automatically Characterizing Large Scale Program Behavior, ” *the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

5. R.E. Wunderlich, T.F. Wenisch, B. Falsafi and J.C. Hoe. "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *International Symposium on Computer Architecture*, 2003
6. R.E. Wunderlich, T.F. Wenisch, B. Falsafi and J.C. Hoe. "An evaluation of Stratified Sampling of Microarchitecture Simulations," in *Workshop on Duplicating, Deconstructing and Debunking*, 2004
7. J. J. Yi, V. Kodakara, R. Sendag, D.J. Lilja and D.M. Hawkins, "Characterizing and Comparing Prevailing Simulation Techniques, " in *International Symposium on High-Performance Computer Architecture*, 2005
8. Y. Luo, J. Yang, L. Bhuyan and L. Zhao, "NePSim: A Network Processor Simulator with Power Evaluation Framework," in *IEEE Micro*, Sept/Oct, 2004
9. Z.X. Tan, C. Lin, H. Yin and B.Li, "Optimization and Benchmark of Cryptographic Algorithms on Network Processors" in *IEEE Micro*, Sept/Oct, 2004
10. P.S.Levy and S. Lemeshow, "Sampling of Populations: Methods and Applications", John Wiley and Sons, Inc., 1999
11. D.Pelleg and A. Moore. "Accelerating Exact K-means Algorithms with Geometric Reasoning," in *Proceedings of the Fifth International Conference on Knowledge Discovery in Databases*, 1999
12. Network traces, http://www.nlanr.net
13. S. M. Ross, "Introduction to Probability and Statistics for Engineers and Scientists," Elsevier Academic Press, 2004.
14. D.A. Menasce, V.A.F. Almeida, "Capacity Planning for Web Services", Prentice Hall, Inc. 2001
15. T.M. Conte, M.A. Hirsch and W.W. Hwu. "Combining trace sampling with single pass methods for efficient cache simulation," *the 1996 International Conference on Computer Design*, 1996.
16. E. Perelman, G. Hamerly and Brad Calder, "Picking Statistically Valid and Early Simulation Points" *International Conference on Parallel Architecture and Compilation Techniques*, 2003.

Power Aware External Bus Arbitration for System-on-a-Chip Embedded Systems

Ke Ning[1,2] and David Kaeli[1]

[1] Northeastern University 360 Huntington Avenue, Boston MA 02115
[2] Analog Devices Inc. 3 Technology Way Norwood MA 02062

Abstract. Power efficiency has become a key design trade-off in embedded system designs. For system-on-a-chip embedded systems, an external bus interconnects embedded processor cores, I/O peripherals, a direct memory access (DMA) controller, and off-chip memory. External memory access activities are a major source of energy consumption in embedded systems, and especially in multimedia platforms. In this paper, we focus on the energy dissipated due to the address, data, and control activity on the external bus and supporting logic. We build our external bus power model on top of a cycle-accurate simulation framework that quantifies the bus power based on memory bus state transitions. We select an Analog Devices ADSP-BF533 multimedia system-on-a-chip embedded system as our target architecture model. Using our power-aware external bus arbitration schemes, we can reduce overall power by as much as 18% in video processing applications, and by 12% on average for the test suites studied. Besides reducing power consumption, we also obtained an average performance speedup of 24% when using our power-aware arbitration schemes.

Keywords: Power-aware, external memory, bus arbitration, embedded systems, media processor.

1 Introduction

Modern embedded systems are becoming increasingly limited by memory performance and system power consumption. The power associated with off-chip accesses can dominate the overall power budget. The memory power/speed problem is even more acute for embedded media processors that possess memory intensive access patterns and require streaming serial memory access that tends to exhibit low temporal locality (i.e., poor data cachablity). Without more effective bus communication strategies, media processors will continue to be limited by memory power and memory performance.

One approach to addressing both issues is to consider how best to schedule off-chip accesses. Due to the intrinsic capacitance of the bus lines, a considerable amount of power is required at the I/O pins of a system-on-a-chip processor when data has to be transmitted through the external bus [1,2]. The capacitance associated with the external bus is much higher than the internal node

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 87–101, 2005.

capacitance inside a microprocessor. For example, a low-power embedded microprocessor system like an Analog Devices ADSP-BF533 running at 500 MHz consumes about 374 mW on average during normal execution. Assuming a 3.65 V voltage supply and a bus frequency of 133 MHz, the average external power consumed is around 170 mW, which accounts for approximately 30% of the overall system power dissipation [3].

In modern CMOS circuit design, the power dissipation of the external bus is directly proportional to the capacitance of the bus and the number of transitions ($1 \rightarrow 0$ or $0 \rightarrow 1$) on bus lines [4,5]. In general, the external bus power can be expressed as:

$$P_{bus} = C_{bus} V_{ext}^2 f k \mu + P_{leakage} \tag{1}$$

In the above equation, C_{bus} denotes the capacitance of each line on the bus, V_{ext} is the bus supply voltage, f is the bus frequency, k is the number of bit toggles per transition on the full width of the bus, and μ is the bus utilization factor. This power equation is an activity-based model. It not only accounts for the dynamic power dissipated on the bus, but includes the pin power that drives the signal I/O's related to external bus communication. $P_{leakage}$ is the power dissipated on the bus due to leakage current.

The techniques to minimize the power dissipation in buses have been well explored in previous research. The main strategies have been to utilize improved bus encodings to minimize the bus activity. Various mixed-bus encoding techniques (e.g., Gray codes and redundant codes) were developed to save on bus power. Gray code addressing is based on the fact that bus values tend to change sequentially and they can be used to switch the least number of signals on the bus.

However, better performance can be obtained by using redundant codes [1]. A number of redundant codes have been proposed that add signals on the bus lines in order to reduce the number of transitions. Bus-invert coding [6]is one class of the redundant codes. Bus-invert coding adds an INV signal on the bus to represent the polarity of the address on the bus. The INV signal value is chosen by considering how best to minimize the hamming distance between the last address on the bus and the current one. Some codes can be applied to both the data and address buses, though some are more appropriate for addresses.

In our previous work, we described a bus modeling system that can capture bus power in the same framework of a cycle-accurate simulator for an embedded media processor [7]. We discussed an initial design of a power-aware bus arbitration scheme. The main contributions of this paper are a completed design of our power-aware bus arbitration scheme that also considers using pipelined SDRAM, and we also consider a broader range of multimedia applications. This paper is organized as follows. In section 2 we describe the target architecture for our work, which contains a system-on-a-chip media processor, SDRAM memory, and an external bus interface unit. We also present our power modeling methodology. Section 3 describes a number of different bus arbitration algorithms that we consider for power and performance optimizations. Section 4 presents power/performance results of MPEG-2, JPEG, and PGP benchmarks

for traditional arbitration schemes and our power-aware schemes. Finally, Section 5 presents conclusions.

2 System-on-a-Chip Architectures

2.1 Interconnect Subsystem

Modern system-on-a-chip embedded media systems include many components: a high-speed processor core, hardware accelerators, a rich set of peripherals, direct memory access (DMA), on-chip cache and off-chip memory. The system architecture considered in our study includes a single core, several peripherals, and off-chip SDRAM memory, and is similar to many current embedded platforms.

For multimedia applications, throughput requirements are increasing faster and faster. Today, for a D1 (720x480) video codec (encoder/decoder) media node, we need to be able to process 10 million pixels per second. This workload requires a media processor for computation, devices to support high speed media streaming and data conversion via a parallel peripheral interface (PPI), and a synchronous serial port (SPORT) for interfacing to high speed telecom interfaces. The high data throughput requirements associated with this platform make it impossible to store all the data in an on-chip memory or cache. Therefore, a typical multimedia embedded system usually provides a high-speed system-on-a-chip microprocessor and a very large off-chip memory. The Analog Devices Blackfin family processors, the Texas Instrument OMAP, and the SigmaDesign EM8400 series are all examples of low-power embedded media chipsets which share many similarities in system design and bus structure. The system architecture assumed in this paper is based on these designs and is shown in Figure 1.

When trying to process streaming data in real-time, the greatest challenge is to provide enough memory bandwidth in order to sustain the necessary data rate. To insure sufficient bandwidth, hardware designers usually provide multiple

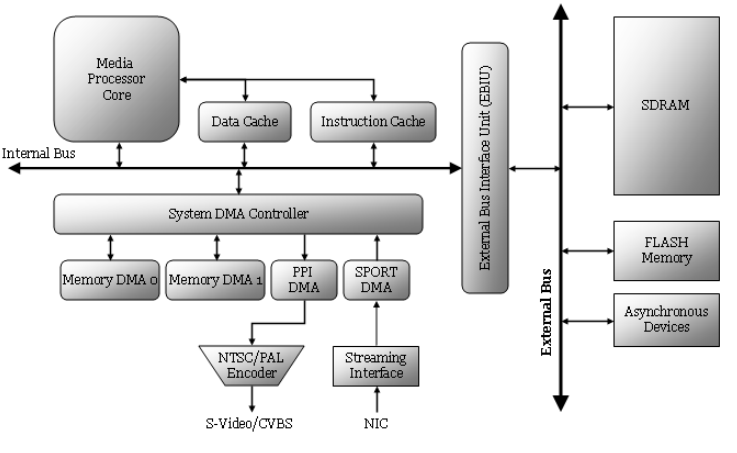


Fig. 1. Our target embedded media system architecture

buses in the system, each possessing different bus speeds and different protocols. An external bus is used to interface to the large off-chip memory system and other asynchronous memory-mapped devices. The external bus has a much longer physical length than other buses, and thus typically has higher bus capacitance and power dissipation. The goal of this work is to accurately model this power dissipation in a complete system power model so we can explore new power-efficient scheduling algorithms for the external memory bus.

2.2 External Bus Interface Unit

In the system design shown in Figure 1, there are two buses, one internal bus and one external bus, These two buses are bridged by an external bus interface unit (EBIU), which provides a glue-less interface to external devices (i.e., SDRAM memory, flash memory and asynchronous devices).

There are two sub-modules inside the EBIU, a bus arbitrator and a memory controller. When the units (processor or DMA's) in the system need to access external memory, they only need to issue a request to the EBIU buffer through the internal bus. The EBIU will read the request and handle the off-chip communication tasks through the external bus. Due to the potential contention between users on the bus, arbitration for the external bus interface is required. The bus arbitrator grants requests based on a pre-defined order. Only one access request can be granted at a time. When a request has been granted, the memory controller will communicate with the off-chip memory directly based on the specific memory type and protocol. The EBIU can support SDRAM, SRAM, ROM, FIFOs, flash memory and ASIC/FPGA designs, while the internal units do not need to discriminate between different memory types. In this paper, we use multi-banked SDRAM as an example memory technology and integrate SDRAM state transitions into our external bus model (our modeling framework allows us to consider different memory technologies, without changing the base system-on-a-chip model).

2.3 Bus Power Model

The external bus power includes dynamic power to charge and discharge the capacitance along the external bus, and the pin power to drive the bus current. The external bus power is highly dependent on the memory technology chosen. In past work on bus power modeling, little attention has been paid to the impact of the chosen memory technology. While we have assumed an SDRAM in our power model in this work, we can use the same approach with other types of memory modules. The external bus power associated with each transaction will be the total number of pins that toggle on the bus. We include in our model the power consumption due to the commands sent on the control bus, the row address and column address on the address bus, and the data on data bus. The corresponding leakage power is also considered in our model.

SDRAM is commonly used in cost-sensitive embedded applications that require large amounts of memory. SDRAM has a three-dimensional structure

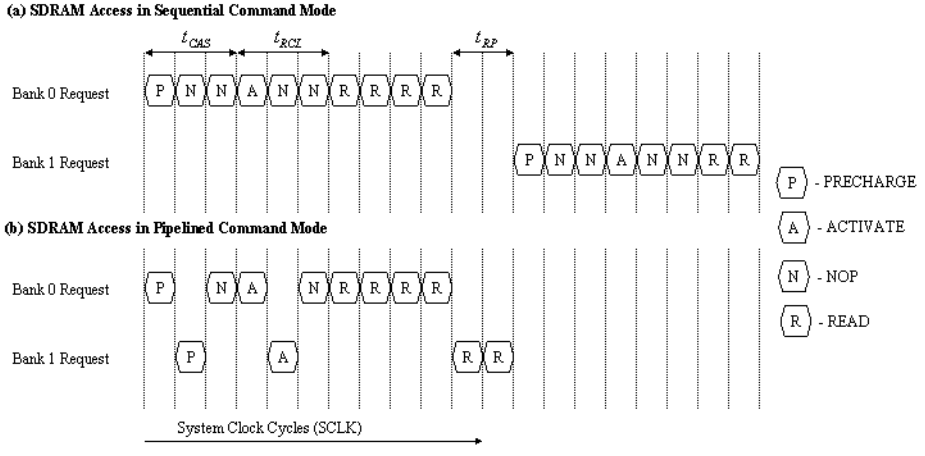


Fig. 2. Timing diagram showing two memory accesses for both sequential and pipelined command SDRAM

model. It is organized in multiple banks. Inside each bank, there are many pages, which are selected by row address. The memory access is on a command-by-command basis. An access involves processing a PRECHARGE and an ACTIVATE command before a physical READ/WRITE command. At the same time of an ACTIVATE and READ/WRITE command, the corresponding row and column addresses are sent on the address bus.

To maximize memory bandwidth, modern SDRAM components allow for pipelining memory commands [8], which eliminates unnecessary stall cycles and NOP commands on the bus. While these features increase the memory bandwidth, they also reduce the bus command power. Consecutive accesses to different rows within one bank have high latency and cannot be pipelined, while consecutive accesses to different rows in different banks can be pipelined. Figure 2 is a timing diagram for processing two read operations in sequential access SDRAM and pipelined access SDRAM.

In our bus model, we assume that the power to drive the control bus and address bus are the similar. For each read/write request, we first determine the series of commands needed to complete that request. For each command, the bus state transitions, the number of pin toggles, and the bus utilization factor is recorded. Finally, the average bus power dissipated is calculated using Equation 1.

3 Bus Arbitration

The bandwidth and latency of external memory system are heavily dependent on the manner in which accesses interact with the three-dimensional SDRAM structure. The bus arbitration unit in the EBIU determines the sequencing of

load/store requests to SDRAM, with the goals of reducing contention and maximizing bus performance. The requests from each unit will be queued in the EBIU's wait queue buffer. When a request is not immediately granted, the request enters stall mode. Each request can be represented as a tuple (t, s, b, l), where t is the arrival time, s identifies the request (load or store), b is the address of the block, and l is the extent of the block. The arbitration algorithm schedules requests sitting in the wait queue buffer with a particular performance goal in mind. The algorithm needs to guarantee that bus starvation will not occur.

3.1 Traditional Algorithms

A number of different arbitration algorithms have been used in microprocessor system bus designs. The simplest algorithm is *First Come First Serve* (FCFS). In this algorithm, requests are granted on the bus based on the order of arrival. This algorithm simply removes contention on the external bus without any optimization and pre-knowledge of the system configuration. Because FCFS schedules the bus naively, the system performs poorly due to instruction and data cache stalls. The priority of cache accesses and DMA access are equal (though cache accesses tend to be more performance critical than DMA accesses). An alternative is to have a *Fixed Priority* scheme where cache accesses are assigned higher priority than DMA accesses. For different DMA accesses, peripheral DMA accesses will have higher priority than memory DMA accesses. This differentiation is needed because if a peripheral device access is held off for a long period of time, it could cause the peripheral to lose data or time out. The Fixed Priority scheme selects the request with highest priority in the waiting queue instead of just selecting the oldest. Using Fixed Priority may provide similar external bus performance as the FCFS algorithm, but the overall system performance should be better if the application is dominated by cache accesses. For real-time embedded applications which are dominated by DMA accesses, cache accesses can be tuned such that cache misses are infrequent. Cache fetches can be controlled to occur only at non-critical times using cache prefetching and locking mechanisms. Therefore, for real-time embedded applications, the FCFS and Fixed Priority schemes produce very similar external bus behavior.

3.2 Power Aware Algorithms

To achieve efficient external bus performance, FCFS and Fixed Priority are not sufficient. Power and speed are two major factors of bus performance. In previous related work, dynamic external bus arbitration and scheduling decisions were primarily driven by bus performance and memory bandwidth [8,9]. If a power-efficient arbitration algorithm is aware of the power and cycle costs associated with each bus request in the queue, each request can be scheduled to achieve more balanced power/performance. The optimization target can be to minimize power P, minimize delay D, or more generally to minimize $P^n D^m$. This problem can be formulated as a shortest Hamiltonian path (SHP) on a properly defined graph. The Hamiltonian path is defined as the path in a directed graph that

visits each vertex exactly once, without any cycles. The shortest Hamiltonian path is the Hamiltonian path that has the minimum weight. The problem is NP-complete, and in practice, heuristic methods are used to solve the problem [10].

Let R_0 denote the most recently serviced request on the external bus. R_1, R_2, ... R_L are the requests in the wait queue. Each request R_i consists of four elements (t_i, s_i, b_i, l_i), representing the arrival time, the access type (load/store), the starting address, and the access length. The bus power and delay are dependent on the current bus state and the following bus state for each request. The current bus state is the state of the bus after the previous bus access has completed. $P(i, j)$ represents the bus power dissipated for request R_j, given R_i was the immediate past request. $D(i, j)$ is the time between when request R_j is issued and when R_j is completed, where R_i was the immediate past request. The cost associated with scheduling request R_j after request R_i can be formulated as $P^n(i, j)D^m(i, j)$. We can define a directed graph $G = (V, E)$ whose vertices are the requests in the wait queue, with vertex 0 representing the last request completed. The edges of the graph include all pairs (i, j). Each edge is assigned a weight $w(i, j)$, and is equal to the power delay product of processing request R_j after request R_i.

$$w(i, j) = P^n(i, j)D^m(i, j), n, m = 0, 1, ... \tag{2}$$

The problem of optimal bus arbitration is equivalent to the problem of finding a Hamiltonian path starting from vertex 0 in graph G with a minimum path traversal weight. Figure 3 describes a case when there are 3 requests are in the wait queue. One of the Hamiltonian paths is illustrated with a dot line. The weight of this path is $w(0, 3) + w(3, 1) + w(1, 2)$. For each iteration, a shortest Hamiltonian path will be computed to produce the minimum weight path. The

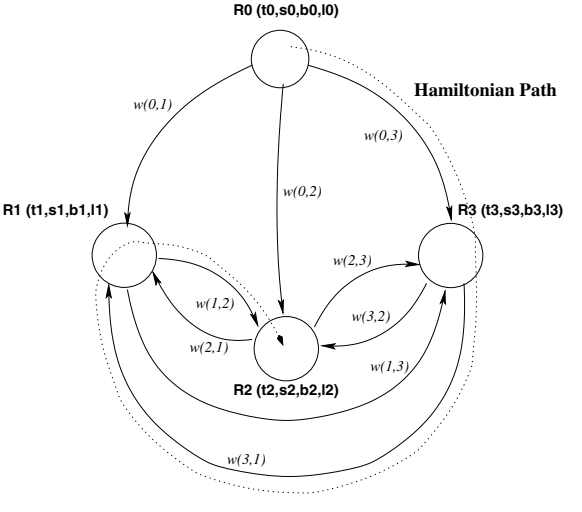


Fig. 3. Hamiltonian Path Graph

first request after request R_0 on that path will be the request selected in next bus cycle. After the next request is completed, a new graph will be constructed and a new minimum Hamiltonian path will be found.

Finding the shortest Hamiltonian path has been shown to be NP-complete. To produce a shortest path, we use heuristics. Whenever the path reaches vertex R_i, the next request R_k with minimum $w(i,k)$ will be chosen. This is a greedy algorithm, which selects the lowest weight for each step. The bus arbitration algorithm only selects the second vertex on that path. We avoid searching the full Hamiltonian path, and so the bus arbitration algorithm can simply select a request based on finding the minimum $w(0,k)$ from request R_0. The complexity of this heuristic is $O(L)$. When $w(i,j) = P(i,j)$, arbitration will try to minimize power. When $w(i,j) = D(i,j)$, then we can minimize delay. To consider the power efficiency, the power delay product can be used. Selecting different values for n and m change how we trade off power with delay using weights $w(i,j)$.

3.3 Target Architecture

In our experimental study, we used a power model of the Analog Devices Blackfin family system-on-a-chip processors as our primary system model. We run code developed for ADSP-BF533 EZ-Kit Lite board using the VisualDSP++ toolset. This board provides a 500 MHz ADSP-BF533 microprocessor, 16 MB of SDRAM, and a CCIR-656 video I/O interface. Inside the ADSP-BF533 microprocessor, there are both L1 instruction and data caches. The instruction cache is 16 KB 4-way set associative. The data cache is 16 KB 2-way set associative. Both caches use a 32 byte cache line size. The SDRAM module selected is the Micron MT48LC16M16A2 16 MB SDRAM. The SDRAM interface connects to 128 Mbit SDRAM devices to form one 16 MB of external memory. The SDRAM is organized in 4 banks, with a 1 KB page size. It also has following characteristics to match the on-chip SDRAM controller specification: 3.3V supply voltage, 133 MHz operating frequency, burst length of 1, column address strobe (CAS) latency t_{CAS} of 3 system clock cycles, t_{RP} and t_{RCD} equal to 2 system clock cycles, refresh rate programmed at 4095 system clock cycles. We used the Analog Devices Blackfin frio-eas-rev0.1.7 toolkit to integrate this model. The power model has been validated with physical measurements as described in [11]. To make sure the arbitration algorithm does not produce long-term starvation, a time-out mechanism was added for the requests. The timeout values for cache and memory DMA are 100 and 550 cycles, respectively.

4 Experimental Setup

4.1 Benchmarks

Experiments were run on a set of multimedia workloads. We chose MPEG-2 for video processing, JPEG for image compression and PGP for cryptography. All

Table 1. Benchmarks

Name	Description
MPEG2-ENC	MPEG-2 Video encoder with 720x480 4:2:0 input frames.
MPEG2-DEC	MPEG-2 Video decoder of 720x480 sequence with 4:2:2 CCIR frame output.
JPEG-ENC	JPEG image encoder for 512x512 image.
JPEG-DEC	JPEG image decoder for 512x512 image.
PGP-ENC	Pretty Good Privacy encryption and digital signature of text message.
PGP-DEC	Pretty Good Privacy decryption of encrypted message.

three benchmark suites are representative and commonly used applications for multimedia processing. MPEG-2 is the dominant standard for high-quality digital video transmission and DVD. We selected real-time MPEG-2 encoder and decoder source codes that include optimized Blackfin MPEG-2 libraries. The input datasets used are the *cheerleader* for encoding (the size is 720x480 and the format is interlaced video) and *tennis* for decoding (this image is encoded by the MPEG-2 reference encoder, the size is also 720x480, and the format is progressive video). Both inputs are commonly used by the commercial multimedia community.

JPEG is a standard lossy compression method for full color images. The JPEG encoder and decoder used also employ optimized Blackfin libraries. The input image is Lena (the size is 512x512 in a 4:2:2 color space).

PGP stands for *Pretty Good Privacy*, and provides for encryption and signing data. The signature we use is a 1024 bit cryptographically-strong one-way hash function of the message (MD5). To encrypt data, PGP uses a block-cipher IDEA and RSA for key management and digital signature.

In order to measure the external bus power and be able to assess the impact of our power-efficient bus arbitration algorithm, we developed the following simulation framework. First, we modified the Blackfin instruction-level simulator to include the system bus model and cache activity. From this model, we feed all accesses generated on the external bus to an EBIU model. The EBIU model faithfully simulates the external bus behavior, capturing detailed SDRAM state transitions and allows us to considered different bus arbitration schemes. The average bus power and performance are computed from the simulation results produced by our integrated simulator.

4.2 Results

There are eleven different bus arbitration schemes evaluated in our simulation environment. We considered two traditional schemes: (1) Fixed Priority (FP), (2) First Come First Serve (FCFS), and 9 different power-aware schemes. For Fixed Priority, we assign the following priority order (from highest to lowest): instruction cache, data cache, PPI DMA, SPORT DMA, memory DMA. In the power-aware schemes, each scheme is represented by the pair of power/delay coefficients (n, m) of the arbitration algorithm. n and m are the exponents shown in Equation 2. Different n and m values will favor either power or delay. (1, 0) is the minimum power scheme, (0, 1) is the minimum delay scheme, and (1, 1), (1, 2), (2, 1), (1, 3), (2, 3), (3, 2), (3, 1) consider a balance between power and

delay by using different optimization weights. We present experimental results for both power and delay. The MPEG-2 encoder and decoder simulation results are shown in Figure 4, JPEG encoder and decoder are shown in Figure 5 and PGP encryptor and decryptor are shown in Figure 6. All the experiments consider both sequential command mode SDRAM and pipelined command mode SDRAM.

In all applications, the power dissipation for the power-aware arbitration schemes is much lower when compared to using Fixed Priority or FCFS. The power-aware schemes also benefit from fewer bus delays. These conclusions are consistent across all of the applications studied and are also consistent when using either sequential command SDRAM or pipelined command SDRAM. In the MPEG-2 case, the power-aware scheme (1, 0) enjoys an 18% power savings relative to a Fixed Priority scheme for encoder and 17% for decoder. The same power-aware scheme also achieved a 40% reduction in cycles when compared to the Fixed Priority scheme on MPEG-2 decoder, and a 10% reduction for MPEG-2 encoder.

To factor out the impact of sequential versus pipelined command mode from the power savings, we show in Table 2 the bus power savings and we show in Table 3 the cycle savings. Inspecting the two tables, we can see that the power-aware arbitration scheme achieves an average power savings of 12% and an average speedup of 24% over all 6 applications. There exist some variations in power savings and speed up achieved. These variations are primarily due to differences in bus utilization across the different applications. For high traffic applications, external memory access requests are more bursty, In those cases, our power-aware schemes provide a larger improvement than in low traffic appli-

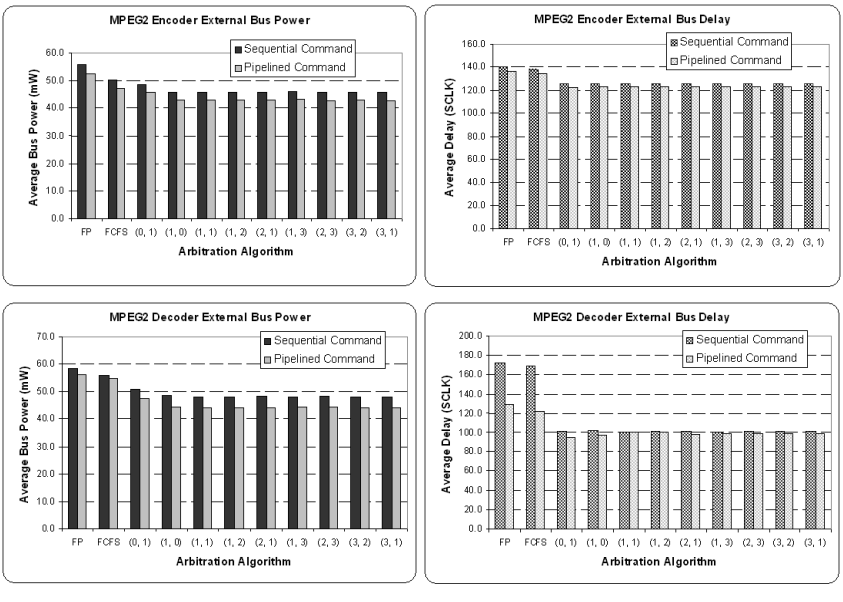


Fig. 4. External bus power/delay results for MPEG-2 video encoder and decoder

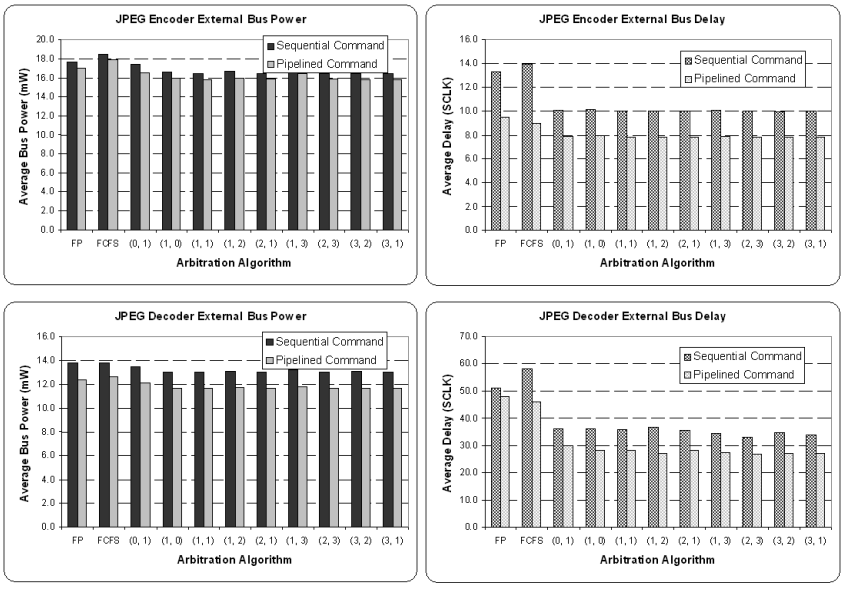


Fig. 5. External bus power/delay for JPEG image encoder and decoder

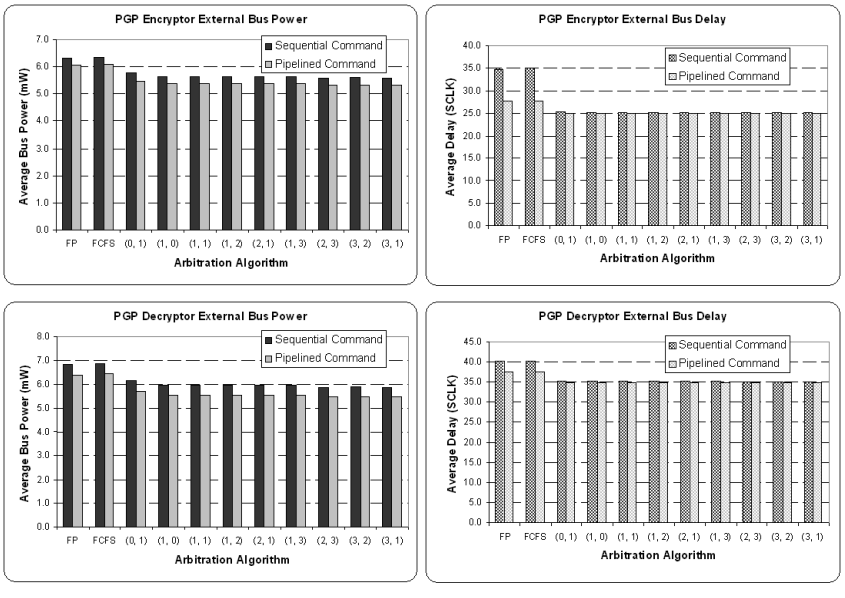


Fig. 6. External bus power/delay for PGP encryption and decryption

Table 2. External bus power savings of (1, 0) arbitration vs. fixed priority arbitration in sequential command SDRAM

	Fixed priority power (mW)	Arbitration (1, 0) Power (mW)	Power savings
MPEG2-ENC	55.85	45.94	18%
MPEG2-DEC	58.47	48.62	17%
JPEG-ENC	17.64	16.57	6%
JPEG-DEC	13.77	12.99	6%
PGP-ENC	6.33	5.66	11%
PGP-DEC	6.81	5.94	13%
Average savings			12%

Table 3. External bus speedup of (1, 0) arbitration vs. fixed priority in sequential command SDRAM

	Fixed priority delay (SCLK)	Arbitration (1, 0) Delay (SCLK)	Speedup
MPEG2-ENC	140.36	126.10	10%
MPEG2-DEC	171.94	101.52	41%
JPEG-ENC	13.30	10.19	23%
JPEG-DEC	51.22	36.04	30%
PGP-ENC	34.87	25.21	28%
PGP-DEC	40.28	35.22	13%
Average Speedup			24%

Table 4. External bus power savings of (1, 0) arbitration vs. fixed priority arbitration in pipelined command SDRAM

	Fixed Priority Power (mW)	Arbitration (1, 0) Power (mW)	Power savings
MPEG2-ENC	52.51	42.84	18%
MPEG2-DEC	56.29	44.58	21%
JPEG-ENC	16.97	15.93	6%
JPEG-DEC	12.41	11.68	6%
PGP-ENC	6.05	5.39	11%
PGP-DEC	6.40	5.54	13%
Average savings			13%

Table 5. External bus speedup of (1, 0) arbitration vs. fixed priority in pipelined command SDRAM

	Fixed priority delay (SCLK)	Arbitration (1, 0) Delay (SCLK)	Speedup
MPEG2-ENC	136.73	122.79	10%
MPEG2-DEC	128.82	97.57	24%
JPEG-ENC	9.50	7.93	16%
JPEG-DEC	48.02	28.20	41%
PGP-ENC	27.61	24.92	10%
PGP-DEC	37.34	34.78	7%
Average Gain			18%

cations, in which the requests are less bursty. The greater the number of requests in the queue, the greater the opportunity that the arbitrator can effect an improvement. Similarly, in Tables 4 and 5, the pipelined command SDRAM obtains on average a 13% power savings and a 18% performance speedup.

From inspecting the results shown in Tables 2-5, we can see that the choice of using sequential versus pipelined command modes is not a factor when con-

Table 6. Power and speed improvements for pipelined vs. sequential command mode SDRAM

	Avg. power in sequential mode (mW)	Avg. power in pipelined mode (mW)	Power reduction of pipelined commands	Avg. delay sequential mode (SCLK)	Avg. delay in pipelined mode (SCLK)	Speedup of in pipelined commands
MPEG2-ENC	47.45	44.40	6%	128.31	124.99	3%
MPEG2-DEC	50.13	46.64	7%	113.38	103.38	9%
JPEG-ENC	16.86	16.26	4%	10.68	8.10	24%
JPEG-DEC	13.21	11.89	10%	38.74	31.28	19%
PGP-ENC	5.78	5.50	5%	26.95	25.40	6%
PGP-DEC	6.11	5.70	7%	36.09	35.25	2%
Avg. speedup			6%			10%

sidering the performance impact of the bus arbitration algorithm. However, the pipelined command mode does decrease the request delay period by overlapping bus command stall cycles with other non-collision producing commands. Pipelining also helps to reduce the bus power by using one command's PRECHARGE or ACTIVATE stall cycles to prepare for the next READ/WRITE command (versus sending NOP commands). Table 6 summarizes the results between the sequential command mode and pipelined command mode SDRAMs. The results show that the pipelined command mode SDRAM can produce a 6% power savings and a 10% speedup.

Comparing the results across the power-efficient schemes, we can see that the performance differences are small, and that no one scheme provides significant advantages over the rest. The scheme (1, 0) (i.e., the minimum power approach) is actually more favorable with regards to design implementation. Scheme (1, 0) basically needs a Hamming distance (XOR) computation unit and a comparator. For each iteration, the arbitrator uses the Hamming distance computation unit to accumulate the power used for each request that is pending in the wait queue, and uses the comparator to select the minimum. For $0.13\mu m$ CMOS technology and a 1.2 V power supply, an XOR transistor takes about 30 fJ to switch the

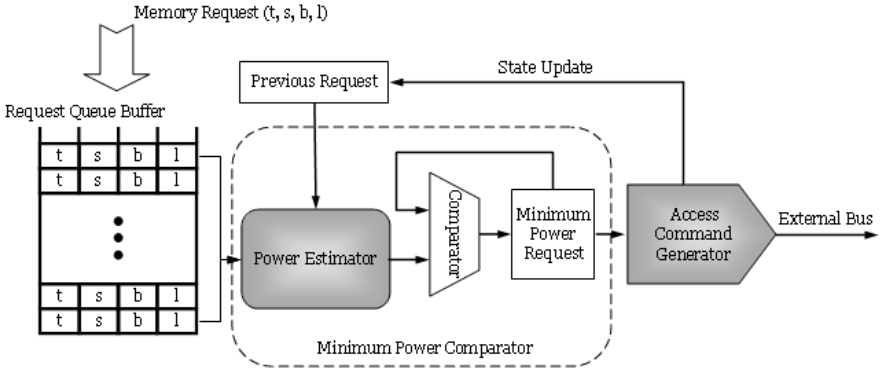


Fig. 7. Our power-aware bus arbitration architecture

transistor state in the slow N and slow P process corner. In our case, the number of transistors to implement the (1, 0) arbitrator is on the order of 10^3.

Figure 7 shows the architecture of the (1, 0) power-aware arbitrator. It contains three major components, a single request queue, a single minimum power comparator, and the memory access command generator. As memory access requests arrive, they allocate storage space while waiting for service in the request queue. The power estimator computes a request's power relative to the previous request. The comparator selects the minimum power request and stores it. When the currently active request finishes, the minimum power request will be sent to the bus by the access command generator, which also updates the previous request register which will be used to perform request selection in the next cycle. Power estimation is performed serially for each request in the queue. All requests will share the same power estimator logic. Using such a shared structure will reduce the hardware complexity, but may introduce some latency. In future work we will investigate how to provide for parallel power estimation that balances latency and power overhead.

5 Conclusions

Memory bandwidth has become a limiting factor in high performance embedded computing. For future multimedia processing systems, bandwidth and power are both critical issues that need to be addressed. This paper proposes a set of new external bus arbitration schemes that balance bus power and delay. Our experiments are based on modeling a low-end embedded multimedia architecture while running six multimedia benchmarks. Our results show that significant power reductions and performance gains can be achieved using power-aware bus arbitration schemes compared to traditional arbitration schemes. We also considered the impact of using both sequential and pipelined SDRAM models. Finally, a hardware implementation of (1, 0) power-aware arbitrator is proposed.

References

1. Benini, L., De Micheli, G., Macii, E., Sciuto, D., Silvano, C.: Address bus encoding techniques for system-level power optimization. In: Proceedings of the Conference on Design, Automation and Test in Europe, IEEE Computer Society (1998) 861–867
2. Panda, P.R., Dutt, N.D.: Reducing address bus transitions for low power memory mapping. In: Proceedings of the 1996 European Conference on Design and Test, IEEE Computer Society (1996) 63
3. Analog Devices Inc. Norwood, MA: Engineer-to-Engineer Note EE-229: Estimating Power for ADSP-BF533 Blackfin Processors (Rev 1.0). (2004)
4. Givargis, T.D., Vahid, F., Henkel, J.: Fast cache and bus power estimation for parameterized system-on-a-chip design. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE), ACM Press (2000) 333–339
5. Sotiriadis, P., Chandrakasan, A.: Low-power bus coding techniques considering inter-wire capacitances. In: Proceedings of IEEE Conference on Custom Integrated Circuits (CICC'00). (2000) 507–510

6. Stan, M., Burleson, W.: Bus-invert coding for low-power I/O. IEEE Transactions on Very Large Scale Integration (VLSI) Systems (1995) 49–58
7. Ning, K., Kaeli, D.: Bus power estimation and power-efficient bus arbitration for system-on-a-chip embedded systems. In: Workshop on Powre-Aware Computer Systems PACS'04, 37th Annual IEEE/ACM International Symposium on Microachitecture (2004)
8. Rixner, S., Dally, W.J., Kapasi, U.J., Mattson, P., Owens, J.D.: Memory access scheduling. In: ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture, New York, NY, USA, ACM Press (2000) 128–138
9. Lyuh, C.G., Kim, T.: Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In: DAC '04: Proceedings of the 41st Annual Conference on Design Automation, New York, NY, USA, ACM Press (2004) 81–86
10. Rubin, F.: A search procedure for hamilton paths and circuits. J. ACM **21** (1974) 576–580
11. VanderSanden, S., Gentile, R., Kaeli, D., Olivadoti, G.: Developing energy-aware strategies for the blackfin processor. In: Proceedings of Annual Workshop on High Performance Embedded Computing, MIT Lincoln Laboratory (2004)

Beyond Basic Region Caching: Specializing Cache Structures for High Performance and Energy Conservation

Michael J. Geiger[1], Sally A. McKee[2], and Gary S. Tyson[3]

[1] Advanced Computer Architecture Lab, The University of Michigan,
Ann Arbor, MI 48109-2122
geigerm@eecs.umich.edu
[2] Computer Systems Lab, Cornell University,
Ithaca, NY 14853-3801
sam@csl.cornell.edu
[3] Department of Computer Science, Florida State University,
Tallahassee, FL 32306-4530
tyson@cs.fsu.edu

Abstract. Increasingly tight energy design goals require processor architects to rethink the organizational structure of microarchitectural resources. In this paper, we examine a new multilateral cache organization that replaces a conventional data cache with a set of smaller region caches that significantly reduces energy consumption with little performance impact. This is achieved by tailoring the cache resources to the specific reference characteristics of each application.

1 Introduction

Energy conservation continues to grow in importance for everything from high performance supercomputers down to embedded systems. Many of the latter must simultaneously deliver both high performance and low energy consumption. In light of these constraints, architects must rethink system design with respect to general versus specific structures. Consider memory hierarchies: the current norm is to use very general cache structures, splitting memory references only according to instructions versus data. Nonetheless, different kinds of data are used in different ways (i.e., exhibiting different locality characteristics), and even a given set of data may exhibit different usage characteristics during different program phases. On-chip caches can consume over 40% of a chip's overall power [1]: as an alternative to general caching designs, further specialization of memory structures to better match usage characteristics of the data they hold can both improve performance and significantly reduce total energy expended.

One form of such heterogeneous memory structures, *region-based caching* [2][3][4], replaces a single unified data cache with multiple caches optimized for global, stack, and heap references; this approach works well precisely because these types of references exhibit different locality characteristics. Furthermore, many applications are dominated by data from a particular region, and thus greater specialization of region structures should allow both quantitative (in terms of performance and

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 102 – 115, 2005.

energy) and qualitative (in terms of security and robustness) improvements in system operation. This approach slightly increases required chip area, but using multiple, smaller, specialized caches that together constitute a given "level" of a traditional cache hierarchy and only routing data to a cache that matches those data's usage characteristics provides many potential benefits: faster access times, lower energy consumption per access, and the ability to turn off structures that are not required for (parts of) a given application.

Given the promise of this general approach, in this paper we first look at the heap cache, the region-based memory structure that most data populate for most applications (in fact, the majority of a unified L1 cache is generally populated by heap data). Furthermore, the heap has always represented the most difficult region of memory to manage well in a cache structure. We propose a simple modification to demonstrate the benefits of further specialization: large and small heap caches. If the application exhibits a small heap footprint, we save energy by using the smaller structure and turn off the larger. For applications with larger footprints, we use both structures, but save energy by keeping highly used "hot" data in the smaller, faster, lower-energy cache. The compiler determines (either through profiled feedback or heuristics) which data belong in which cache, and it conveys this information to the architecture via two different `malloc()` functions that allocate data structures in two disparate regions of memory. This allows the microarchitecture to determine what data are to be cached where without the need for additional bits in memory reference instructions and without complex coherence mechanisms.

The architectural approach we describe above addresses one kind of energy consumption—dynamic or switching energy—via smaller caches that reduce the cost of each data access. The second kind of energy consumption that power-efficient caching structures must address is static or leakage energy; for this, we add *drowsy caching* [5][6][7], an architectural technique exploiting dynamic voltage scaling. Reducing supply voltage to inactive lines lowers their static power dissipation. When a drowsy line is accessed, the supply voltage must be returned to its original value before the data may be accessed. Drowsy caches save less power than many other leakage reduction techniques, but do not suffer the dramatically increased latencies of other methods.

Using the MiBench suite [8], we study application data usage properties and the design space for split heap caches. The contributions of this paper are:

- We perform a detailed analysis of heap data characteristics to determine the best heap caching strategy and necessary cache size for each application.
- We show that a significant number of embedded applications do not require a large heap cache and demonstrate significant energy savings with a minimal performance loss for those applications by using a smaller cache. We show energy savings of up to 79% using non-drowsy reduced heap caches and up to 84% using drowsy reduced heap caches.
- For applications that do have a large heap footprint and require a bigger cache, we demonstrate that we can still achieve significant energy savings by identifying a subset of data responsible for the majority of accesses to the heap region and splitting the heap cache into two structures, a small cache for hot data and a large cache for the remaining data. We show energy savings of up to 45% using non-drowsy split-heap caches and up to 78% using drowsy split heap caches.

The remainder of this paper is organized as follows. In Section 2, we present related work on energy-saving techniques in caches, focusing on drowsy and region-based caching and the combination of the two to reduce both dynamic and static energy. Section 3 discusses the caching requirements of heap data, focusing on the footprint size and locality of the heap region. Section 4 describes our experimental setup and presents our results. Section 5 offers conclusions and directions for future work.

2 Related Work

Since dissipated power per access is proportional to cache size, partitioning techniques reduce power by accessing smaller structures. Cache partitioning schemes may be vertical or horizontal. Vertical partitioning adds a level between the L1 and the processor; examples include line buffers [9][10] and filter caches [11]. These structures provide low-power accesses for data with temporal locality, but typically incur many misses and therefore increase average observed L1 latency. Horizontal partitioning divides entities such as cache lines into smaller segments, as in cache subbanking [9][10]. Memory references are routed to the proper segment, reducing dynamic power per data access.

Fig. 1 illustrates a simple example of region on-based caching [3][4], a horizontal partitioning scheme that replaces a unified data cache with heterogeneous caches optimized for global, stack, and heap references. Any non-global, non-stack reference is directed to a normal L1 cache, but since most non-global, non-stack references are to heap data, (with a small number of accesses to text and read-only regions), this L1 is referred to as the *heap cache*. On a memory reference, only the appropriate region cache is activated and draws power. Relatively small working sets for stack and global regions allow their caches to be small, dissipating even less power on hits. Splitting references among caches eliminates inter-region conflicts, thus each cache may implement lower associativity, reducing complexity and access time.

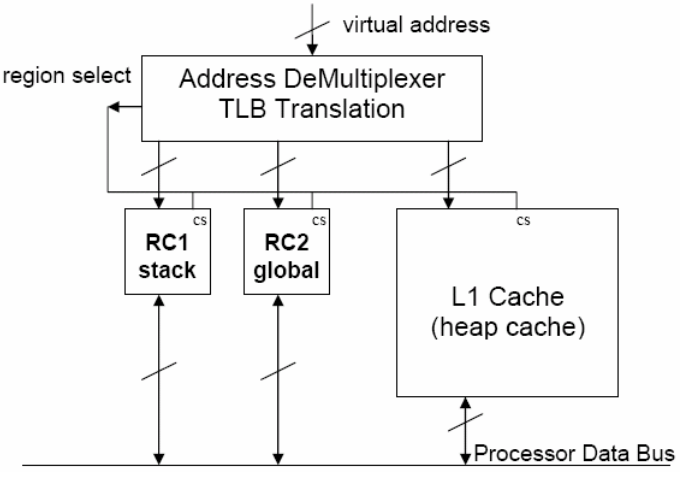


Fig. 1. Memory design for region-based caching (from Lee [4])

The downside to region-based caching is that increased cache capacity leads to higher static energy dissipation. Drowsy region-based caching [2] attacks this problem by implementing drowsy caching 5[5][6][7], a leakage energy reduction technique, within the region-based caches. In a drowsy cache, inactive lines use a reduced supply voltage; an access to a drowsy line must wait for the supply voltage to return to its full value. After a given interval, lines may switch state from active to drowsy; depending on the policy, lines accessed within the interval may remain active. Geiger et al [2] show that the combination of drowsy and region-based caching yields more benefits than either alone because each technique improves the performance of the other. Drowsy caching all but eliminates the static power increase due to additional region caches, while the partitioning strategy used in region-based caching allows for more aggressive drowsy policies. The drowsy interval of each region cache can be tuned according to the reference characteristics of that region, allowing highly active regions to be less drowsy than inactive regions.

Geiger et al [2] first discuss the idea of split heap caches. They note that the ideal drowsy interval for the heap cache is the same as that of the stack cache, a result that implies at least some of the heap data has locality similar to that of stack data. The authors observe that moving the high-locality data to a separate structure would allow more aggressive drowsy caching of the low-locality data, further reducing the static energy consumption of region-based caches. Using a 4KB cache for hot heap data and maintaining a 32KB cache for low-locality heap data, they show an average energy reduction of 71.7% over a unified drowsy L1 data cache.

3 Characteristics of Heap Data

In this section, we analyze the characteristics of heap cache accesses in applications from the MiBench [8] benchmark suite to determine the best heap caching strategy for each program. We begin by assessing the significance of the heap region within each target application, looking at its overall size and number of accesses relative to the other semantic regions. This information is provided in Table 1. The second and third columns of the table show the number of unique block addresses accessed in the heap cache and the number of accesses to those addresses, respectively. Since our simulations assume 32-byte cache blocks, 1 KB of data contains 64 unique block addresses. The fourth and fifth columns show this same data as a percentage of the corresponding values for all regions (i.e., the fourth column shows the ratio of unique data addresses in the heap region to all unique data addresses in the application). We can see several cases that bear out the previous assertions about heap data: they have a large footprint and low locality. In these applications, the heap cache accesses occupy a much larger percentage of the overall footprint than of the total accesses. The most extreme cases are applications such as *FFT.inverse* and *patricia* in which heap accesses account for over 99% of the unique addresses accessed throughout the programs but comprise less than 7% of the total data accesses. This relationship holds in most applications; heap accesses cover an average of 65.67% of the unique block addresses and account for 29.81% of the total data accesses. In some cases, we see a correlation between footprint size and number of accesses—applications with few heap lines and few accesses, like *pgp.encode*, and applications with a large percentage of both cache lines and accesses,

like *tiff2rgba*. A few outliers buck the trend entirely, containing frequently accessed heap data with a relatively small footprint; *dijkstra* is one example.

We see that about half of the applications have a fairly small number of lines in the heap, with 16 of the 34 applications containing fewer than 1000 unique addresses. The *adpcm* application has the smallest footprint, using 69 and 68 unique addresses in the encode and decode phases, respectively. The typical 32 KB L1 heap cache is likely far larger than these applications need; if we use a smaller heap cache, we can dissipate less dynamic power per access with a minimal effect on performance. Since heap cache accesses still comprise a significant percentage of the overall data accesses, this change should have a noticeable effect on the dynamic energy consumption of these benchmarks. Shrinking the heap cache will also reduce its static energy consumption. Previous resizable caches disable unused ways [12][13] or sets [13][14] in set-associative caches; we can use similar logic to simply disable the entire large heap cache and route all accesses to the small cache when appropriate. In Section 4, we show the effects of this modification on energy and performance.

Shrinking the heap cache may reduce the energy consumption of the remaining benchmarks, but the resulting performance loss may be too great to tolerate for applications with a large heap footprint. However, we can still gain some benefit by

Table 1. Characteristics of heap cache accesses in MiBench applications

Benchmark	# unique addresses	Accesses to heap cache	% total unique addresses	% total accesses
adpcm.encode	69	39971743	27.60%	39.88%
adpcm.decode	68	39971781	26.98%	39.88%
basicmath	252	49181748	61.17%	4.52%
blowfish.decode	213	39190633	39.01%	10.17%
blowfish.encode	212	39190621	38.90%	10.17%
bitcount	112	12377683	42.75%	6.75%
jpeg.encode	26012	10214537	99.16%	29.35%
CRC32	90	159955061	41.10%	16.69%
dijkstra	347	44917851	19.74%	38.31%
jpeg.decode	1510	7036942	90.20%	62.91%
FFT	16629	15262360	99.16%	8.56%
FFT.inverse	16630	14013100	99.17%	6.29%
ghostscript	59594	56805375	97.97%	15.29%
ispell	13286	28000346	96.45%	6.43%
mad	2123	40545761	82.25%	36.41%
patricia	110010	16900929	99.86%	6.59%
pgp.encode	298	252620	7.44%	1.93%
pgp.decode	738	425414	44.92%	1.50%
quicksort	62770	152206224	66.67%	12.89%
rijndael.decode	229	37374614	30.99%	21.70%
rijndael.encode	236	35791440	40.00%	19.62%
rsynth	143825	104084186	99.23%	21.43%
stringsearch	203	90920	18.17%	6.21%
sha	90	263617	20.93%	0.72%
susan.corners	18479	9614163	97.07%	63.56%
susan.edges	21028	22090676	99.12%	62.28%
susan.smoothing	7507	179696772	97.03%	41.72%
tiff2bw	2259	57427236	92.09%	98.50%
tiffdither	1602	162086279	83.09%	62.82%
tiffmedian	4867	165489090	53.03%	79.82%
tiff2rgba	1191987	81257094	99.99%	98.51%
gsm.encode	302	157036702	68.02%	11.66%
typeset	168075	153470300	97.97%	49.00%
gsm.decode	285	78866326	55.56%	21.50%
		AVERAGE	65.67%	**29.81%**

identifying a small subset of addresses with good locality and routing their accesses to a smaller structure. Because we want the majority of references to dissipate less power, we should choose the most frequently accessed lines. The access count gives some sense of the degree of temporal locality for a given address.

Table 2. Number of unique addresses required to cover different fractions of accesses to the heap cache in MiBench applications

Benchmark	# unique addresses	% unique addresses needed to cover given percentage of heap cache accesses				
		50%	75%	90%	95%	99%
adpcm.encode	69	1.45%	2.90%	2.90%	2.90%	2.90%
adpcm.decode	68	1.47%	1.47%	1.47%	1.47%	1.47%
basicmath	252	3.97%	25.40%	48.02%	55.56%	61.90%
blowfish.decode	213	0.94%	1.41%	2.35%	26.76%	55.87%
blowfish.encode	212	0.94%	1.42%	2.36%	26.89%	56.13%
bitcount	112	0.89%	1.79%	2.68%	3.57%	3.57%
jpeg.encode	26012	0.10%	0.65%	2.91%	38.19%	87.28%
CRC32	90	2.22%	3.33%	4.44%	4.44%	4.44%
dijkstra	347	0.29%	18.16%	39.19%	49.57%	63.11%
jpeg.decode	1510	4.77%	12.32%	31.85%	44.11%	59.47%
FFT	16629	0.05%	0.14%	4.82%	40.67%	85.33%
FFT.inverse	16630	0.05%	0.14%	13.02%	44.00%	86.51%
ghostscript	59594	0.01%	0.04%	0.56%	6.64%	57.49%
ispell	13286	0.09%	0.23%	0.46%	0.68%	1.29%
mad	2123	1.32%	2.64%	9.70%	14.88%	24.54%
patricia	110010	0.02%	0.06%	0.32%	36.64%	86.03%
pgp.encode	298	0.67%	1.01%	3.69%	6.71%	26.85%
pgp.decode	738	0.27%	0.41%	1.08%	2.30%	29.67%
quicksort	62770	0.02%	0.04%	0.15%	22.08%	49.13%
rijndael.decode	229	1.31%	2.18%	6.55%	31.44%	57.21%
rijndael.encode	236	1.27%	2.97%	7.63%	32.63%	56.78%
rsynth	143825	0.00%	0.00%	0.01%	1.28%	77.33%
stringsearch	203	17.24%	42.86%	59.61%	65.52%	72.91%
sha	90	1.11%	2.22%	3.33%	3.33%	8.89%
susan.corners	18479	0.03%	3.02%	11.04%	14.87%	32.66%
susan.edges	21028	0.02%	4.92%	15.13%	20.22%	30.42%
susan.smoothing	7507	0.01%	0.09%	13.72%	30.25%	44.11%
tiff2bw	2259	10.27%	15.41%	24.26%	29.39%	37.05%
tiffdither	1602	9.43%	19.60%	25.72%	29.59%	40.76%
tiffmedian	4867	4.03%	10.89%	16.72%	20.81%	47.83%
tiff2rgba	1191987	0.04%	0.11%	57.39%	78.69%	95.73%
gsm.encode	302	2.32%	3.97%	5.96%	7.62%	10.60%
typeset	168075	5.55%	15.41%	25.53%	33.02%	60.12%
gsm.decode	285	0.70%	1.40%	4.21%	5.96%	30.53%
AVERAGE (all apps)		2.14%	5.84%	13.20%	24.49%	45.47%
AVERAGE (>1k unique addrs)		1.99%	4.76%	14.07%	28.11%	55.73%

Usually, a small number of blocks are responsible for the majority of the heap accesses, as shown in Table 2. The table gives the number of lines needed to cover different percentages—50%, 75%, 90%, 95%, and 99%—of the total accesses to the heap cache. We can see that, on average, just 2.14% of the cache lines cover 50% of the accesses. Although the rate of coverage decreases somewhat as you add more blocks—in other words, the first N blocks account for more accesses than the next N blocks—we still only need 5.84% to cover 75% of the accesses, 13.2% to cover 90% of the accesses, 24.49% to cover 95% of the accesses, and 45.47% to cover 99% of the accesses. The percentages do not tell the whole story, as the footprint sizesare wildly disparate for these applications. However, the table also shows that in

applications with large footprints (defined as footprints of 1000 unique addresses or more), the percentage of addresses is lower for the first two coverage points (50% and 75%). This statistic implies that we can identify a relatively small subset of frequently accessed lines for all applications, regardless of overall footprint size.

Since a small number of addresses account for a significant portion of the heap cache accesses, we can route these frequently accessed data to a smaller structure to reduce the energy consumption of the L1 data cache. Our goal is to maximize the low-power accesses without a large performance penalty, so we need to judiciously choose which data to place in the hot heap cache. To estimate performance impact, we use the Cheetah cache simulator [15] to find a lower bound on the miss rate for a given number of input data lines. We simulate fully-associative 2 KB, 4 KB, and 8 KB caches with optimal replacement [16] and route the N most frequently accessed lines to the cache, varying N by powers of 2. We use optimal replacement to minimize conflict misses and give a sense of when the cache is filled to capacity; the actual miss rate for our direct-mapped hot heap cache will be higher.

Tables 3, 4, and 5 show the results of these simulations for 2 KB, 4 KB, and 8 KB caches, respectively. We present only a subset of the applications, omitting programs with small heap footprints and a worst-case miss rate less than 1% because they will perform well at any cache size. These tables show a couple of clear trends. The first is that the miss rate rises precipitously for small values of N, but levels off around N = 512 or 1024 in most cases. This result reflects the fact that the majority of accesses are concentrated at a small number of addresses. The second is that the miss rates remain tolerable for all applications for N values up to 256, regardless of cache size. In order to gain the maximum benefit from split heap caching, we would like to route as many accesses as possible to a small cache. These simulations indicate that varying the cache size will not have a dramatic effect on performance, so we choose the

Table 3. Miss rates for a fully-associative 2 KB cache using optimal replacement for different numbers of input addresses. Applications shown either have a large heap footprint or a worst-case miss rate above 1%.

Benchmark	Miss rate for given N value						
	128	256	512	1024	2048	4096	8192
jpeg.encode	0.2%	0.8%	1.8%	2.5%	2.5%	2.5%	2.5%
dijkstra	4.2%	8.0%	8.0%	8.0%	8.0%	8.0%	8.0%
jpeg.decode	0.4%	0.9%	1.7%	2.8%	2.8%	2.8%	2.8%
FFT	0.1%	0.1%	0.2%	0.3%	0.4%	0.8%	1.4%
FFT.inverse	0.1%	0.1%	0.2%	0.3%	0.5%	0.8%	1.5%
ghostscript	0.0%	0.2%	0.3%	0.5%	0.6%	0.6%	0.8%
ispell	0.2%	0.4%	0.4%	0.4%	0.4%	0.4%	0.4%
mad	0.7%	1.6%	2.4%	2.4%	2.4%	2.4%	2.4%
patricia	0.7%	1.3%	1.8%	1.9%	2.0%	2.0%	2.1%
quicksort	0.0%	0.0%	0.1%	0.1%	0.1%	0.2%	0.2%
rsynth	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%
stringsearch	1.8%	2.0%	2.0%	2.0%	2.0%	2.0%	2.0%
susan.corners	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.2%
susan.edges	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.2%
susan.smoothing	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
tiff2bw	2.5%	3.8%	4.7%	5.7%	5.7%	5.7%	5.7%
tiffdither	0.4%	0.8%	1.3%	1.6%	1.6%	1.6%	1.6%
tiffmedian	0.5%	1.2%	2.0%	3.4%	3.5%	3.4%	3.4%
tiff2rgba	2.5%	3.8%	4.6%	6.1%	7.1%	7.1%	7.1%
typeset	1.4%	2.6%	2.7%	3.0%	3.4%	4.0%	5.0%

Table 4. Miss rates for a fully-associative 4 KB cache using optimal replacement for different numbers of input addresses. Applications are the same set shown in Table 3.

Benchmark	Miss rate for given N value						
	128	256	512	1024	2048	4096	8192
jpeg.encode	0.0%	0.3%	0.9%	1.4%	1.5%	1.5%	1.5%
dijkstra	0.0%	2.7%	2.7%	2.7%	2.7%	2.7%	2.7%
jpeg.decode	0.0%	0.3%	0.7%	1.4%	1.5%	1.5%	1.5%
FFT	0.0%	0.0%	0.1%	0.1%	0.3%	0.6%	1.3%
FFT.inverse	0.0%	0.0%	0.1%	0.2%	0.4%	0.7%	1.4%
ghostscript	0.0%	0.0%	0.0%	0.1%	0.2%	0.3%	0.4%
ispell	0.0%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%
mad	0.0%	0.8%	1.6%	1.6%	1.6%	1.6%	1.6%
patricia	0.0%	0.3%	0.5%	0.6%	0.6%	0.6%	0.7%
quicksort	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%	0.2%
rsynth	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
stringsearch	0.2%	0.5%	0.5%	0.5%	0.5%	0.5%	0.5%
susan.corners	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
susan.edges	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
susan.smoothing	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
tiff2bw	0.0%	2.5%	3.9%	5.0%	5.0%	5.0%	5.0%
tiffdither	0.0%	0.5%	1.1%	1.3%	1.3%	1.3%	1.3%
tiffmedian	0.0%	0.8%	1.3%	2.9%	3.0%	3.0%	3.0%
tiff2rgba	0.0%	2.5%	3.1%	4.6%	5.8%	5.8%	5.8%
typeset	0.0%	0.1%	0.2%	0.5%	0.9%	1.4%	2.3%

Table 5. Miss rates for a fully-associative 8 KB cache using optimal replacement for different numbers of input addresses. Applications shown are the same set shown in Table 3.

Benchmark	Miss rate for given N value						
	128	256	512	1024	2048	4096	8192
jpeg.encode	0.0%	0.0%	0.2%	0.6%	0.6%	0.7%	0.7%
dijkstra	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
jpeg.decode	0.0%	0.0%	0.2%	0.7%	0.7%	0.7%	0.7%
FFT	0.0%	0.0%	0.0%	0.1%	0.3%	0.6%	1.2%
FFT.inverse	0.0%	0.0%	0.0%	0.1%	0.3%	0.7%	1.4%
ghostscript	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.2%
ispell	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
mad	0.0%	0.0%	0.8%	0.9%	0.9%	0.9%	0.9%
patricia	0.0%	0.0%	0.1%	0.2%	0.3%	0.3%	0.3%
quicksort	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%
rsynth	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
stringsearch	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%
susan.corners	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
susan.edges	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
susan.smoothing	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
tiff2bw	0.0%	0.0%	2.4%	3.6%	3.7%	3.7%	3.7%
tiffdither	0.0%	0.0%	0.6%	0.8%	0.8%	0.8%	0.8%
tiffmedian	0.0%	0.0%	0.2%	1.9%	2.0%	2.0%	2.0%
tiff2rgba	0.0%	0.0%	0.5%	1.7%	3.2%	3.3%	3.3%
typeset	0.0%	0.0%	0.0%	0.0%	0.2%	0.6%	1.3%

smallest cache size studied—2 KB—and route the 256 most accessed lines to that cache when splitting the heap. This approach should give us a significant energy reduction without compromising performance.

This approach for determining what data is routed to the small cache does require some refinement. In practice, the compiler would use a profiling run of the application to determine the appropriate caching strategy, applying a well-defined heuristic to the profiling data. We use a simple heuristic in this work to show the potential effectiveness of our caching strategies; a more refined method would likely yield better results.

4 Experiments

The previous section motivates the need for two separate heap caches, one large and one small, to accommodate the needs of all applications. As shown in Table 1, many applications have small heap footprints and therefore do not require a large heap cache; in these cases, we can disable the large cache and place all heap data in the smaller structure. This approach will reduce dynamic energy by routing accesses to a smaller structure and reduce static energy by decreasing the active cache area. Applications with large heap footprints are more likely to require both caches to maintain performance. We showed in Table 2 that most heap references access a small subset of the data; by keeping this hot data in the smaller structure, we can save dynamic energy. In all cases, we can further lower static energy consumption by making the caches drowsy.

Our simulations use a modified version of the SimpleScalar ARM target [17]. We use Wattch [18] for dynamic power modeling and Zhang et al.'s HotLeakage [19] for static power modeling. HotLeakage contains a detailed drowsy cache model, which was used in [20] to compare state-preserving and non-state-preserving techniques for leakage control. HotLeakage tracks the number of lines in both active and drowsy modes and calculates leakage power appropriately. It also models the power of the additional hardware required to support drowsy caching. All simulations use an in-order processor model similar to the Intel StrongARM SA-110 [1].

Table 6 shows simulation results for region-based caches using three different heap cache configurations: a large (32KB) unified heap cache, a small (2KB) unified heap cache, and a split heap cache using both the large and small caches. We present normalized energy and performance numbers, using a single 32KB direct-mapped L1 data cache as the baseline. Because all region-based caches are direct-mapped to minimize energy consumption, we use a direct-mapped baseline to ensure a fair comparison. The shaded cells indicate the best heap caching method for each application. To choose the most effective configuration, we calculate the energy-delay product ratio [21] for each configuration, using the cache organization that yields the lowest value.

As expected, using the small heap cache and disabling the large offers the best energy savings across the board. Most applications consume over 70% less energy in this case; the only exceptions are the *susan.corners* and *susan.edges* applications, which suffer the worst performance losses of any applications under this configuration. 18 of the 34 applications in the MiBench suite experience performance losses of less than 1%, including *ghostscript*, *mad*, *patricia*, *rsynth*, and *susan.smoothing*—all applications with large heap footprints. This result suggests that heap data in theseapplications have good locality characteristics and are frequently accessed while present in the cache. Another application, *quicksort*, suffers significant performance losses for all configurations due to an increased number of stack misses, and therefore still benefits most from using the small heap cache. In all of these cases, we gain substantial energy savings with virtually no performance loss, reducing overall energy consumption by up to 79%. Several applications actually experience small speedups, as indicated by the negative values for performance loss. These speedups result from reduced conflict between regions.

For those applications that suffer substantial performance losses with the small cache alone, the split heap cache offers a higher-performance alternative that still

Table 6. Energy and performance results for various non-drowsy heap caching configurations: a large unified heap cache, a small unified heap cache, and a split heap cache employing both large and small caches. The baseline is a 32KB direct-mapped unified L1 data cache. Shaded cells indicate the most effective heap caching method for each application.

Benchmark	32KB heap cache		2KB heap cache		Split heap cache (2KB hot/32KB cold)	
	Normalized total energy	Execution time increase	Normalized total energy	Execution time increase	Normalized total energy	Execution time increase
adpcm.encode	0.92	0.40%	0.25	0.40%	0.80	0.40%
adpcm.decode	0.89	0.26%	0.25	0.26%	0.75	0.26%
basicmath	0.64	0.90%	0.25	1.08%	0.64	1.08%
blowfish.decode	0.62	0.39%	0.24	0.40%	0.58	0.40%
blowfish.encode	0.62	0.40%	0.24	0.41%	0.58	0.41%
bitcount	0.75	0.00%	0.26	0.00%	0.75	0.00%
jpeg.encode	0.77	-0.13%	0.25	4.89%	0.70	3.66%
CRC32	0.65	-0.38%	0.24	-0.38%	0.57	-0.38%
dijkstra	0.78	0.58%	0.24	6.84%	0.62	6.84%
jpeg.decode	0.88	-0.12%	0.23	11.70%	0.65	9.99%
FFT	0.72	7.32%	0.29	32.26%	0.71	7.64%
FFT.inverse	0.69	5.11%	0.28	22.45%	0.69	5.33%
ghostscript	0.65	-0.21%	0.24	0.11%	0.60	-0.11%
ispell	0.62	-0.03%	0.25	2.83%	0.61	0.36%
mad	0.77	-0.26%	0.23	0.28%	0.61	0.10%
patricia	0.68	0.20%	0.25	0.63%	0.68	0.57%
pgp.encode	0.63	0.04%	0.25	0.04%	0.64	0.04%
pgp.decode	0.62	0.00%	0.25	0.02%	0.64	0.00%
quicksort	0.93	22.47%	0.29	21.42%	0.92	22.62%
rijndael.decode	0.66	0.94%	0.24	1.29%	0.55	1.29%
rijndael.encode	0.64	0.49%	0.24	0.84%	0.55	0.84%
rsynth	0.66	0.73%	0.24	0.97%	0.56	0.92%
stringsearch	0.68	-0.09%	0.25	0.03%	0.68	0.04%
sha	0.66	0.09%	0.26	0.09%	0.69	0.09%
susan.corners	0.90	-2.21%	0.50	250.85%	0.73	-2.14%
susan.edges	0.90	-0.98%	0.35	115.70%	0.76	-0.89%
susan.smoothing	0.78	-0.04%	0.23	0.67%	0.63	0.00%
tiff2bw	1.09	-0.04%	0.21	2.11%	0.83	0.86%
tiffdither	0.93	-0.19%	0.24	7.19%	0.75	1.51%
tiffmedian	1.00	2.88%	0.22	4.54%	0.79	3.17%
tiff2rgba	1.08	-0.60%	0.24	32.04%	0.89	-0.33%
gsm.encode	0.62	0.00%	0.24	0.03%	0.58	0.03%
typeset	0.80	-0.16%	0.23	4.78%	0.75	0.56%
gsm.decode	0.76	0.01%	0.25	0.04%	0.68	0.04%

saves some energy. The most dramatic improvements can be seen in *susan.corners* and *susan.edges*. With the large heap cache disabled, these two applications see their runtime more than double; with a split heap cache, they experience small speedups. Other applications, such as *FFT* and *tiff2rgba*, run over 30% slower with the small cache alone and appear to be candidates for a split heap cache. However, the energy required to keep the large cache active overwhelms the performance benefit of splitting the heap, leading to a higher energy-delay product.

Table 7 shows simulation results for drowsy heap caching configurations. In all cases, we use the ideal drowsy intervals derived in [2]—for the unified heap caches, 512 cycles; for the split heap cache, 512 cycles for the hot heap cache and 1 cycle for the cold heap cache. The stack and global caches use 512 and 256 cycle windows, respectively. Note that drowsy caching alone offers a 35% energy reduction over a non-drowsy unified cache for this set of benchmarks [2].

Table 7. Energy and performance results for various drowsy heap caching configurations: a large unified heap cache, a small unified heap cache, and a split heap cache employing both large and small caches. The baseline is a 32KB direct-mapped unified L1 data cache with a 512 cycle drowsy interval. Shaded cells indicate the most effective heap caching method for each application.

Benchmark	32KB heap cache		2KB heap cache		Split heap cache (2KB hot/32KB cold)	
	Normalized total energy	Execution time increase	Normalized total energy	Execution time increase	Normalized total energy	Execution time increase
adpcm.encode	0.58	0.79%	0.21	0.79%	0.26	0.45%
adpcm.decode	0.57	0.65%	0.21	0.65%	0.25	0.48%
basicmath	0.29	1.01%	0.23	1.20%	0.26	1.11%
blowfish.decode	0.33	0.44%	0.23	0.45%	0.24	0.20%
blowfish.encode	0.33	0.47%	0.23	0.48%	0.24	0.23%
bitcount	0.33	0.00%	0.23	0.00%	0.27	-0.04%
jpeg.encode	0.48	-0.04%	0.21	4.83%	0.29	3.59%
CRC32	0.38	-0.36%	0.22	-0.36%	0.24	-0.73%
dijkstra	0.55	0.82%	0.21	6.91%	0.24	5.99%
jpeg.decode	0.73	-0.10%	0.19	11.70%	0.31	9.76%
FFT	0.33	7.27%	0.23	31.99%	0.27	7.35%
FFT.inverse	0.31	5.08%	0.23	22.28%	0.27	5.11%
ghostscript	0.37	-0.11%	0.22	0.22%	0.26	-0.40%
ispell	0.31	0.01%	0.23	2.84%	0.25	0.19%
mad	0.53	-0.11%	0.21	0.47%	0.26	0.09%
patricia	0.32	0.23%	0.23	0.78%	0.27	0.62%
pgp.encode	0.27	0.04%	0.23	0.04%	0.26	0.00%
pgp.decode	0.27	0.00%	0.23	0.02%	0.26	-0.03%
quicksort	0.39	22.43%	0.23	21.38%	0.29	22.48%
rijndael.decode	0.42	1.52%	0.22	1.99%	0.24	1.81%
rijndael.encode	0.40	0.98%	0.22	1.50%	0.24	1.29%
rsynth	0.41	0.87%	0.22	1.13%	0.25	0.65%
stringsearch	0.31	0.10%	0.23	0.22%	0.26	0.20%
sha	0.27	0.14%	0.23	0.14%	0.26	0.14%
susan.corners	0.73	-2.21%	0.22	250.32%	0.45	-1.96%
susan.edges	0.73	-0.98%	0.20	115.61%	0.48	-0.67%
susan.smoothing	0.57	-0.04%	0.20	0.76%	0.33	-0.39%
tiff2bw	1.00	-0.05%	0.16	2.78%	0.55	1.15%
tiffdither	0.73	0.11%	0.19	7.44%	0.39	2.43%
tiffmedian	0.86	2.91%	0.17	4.99%	0.50	3.36%
tiff2rgba	1.00	-0.60%	0.16	32.04%	0.68	0.05%
gsm.encode	0.34	0.01%	0.22	0.05%	0.24	-0.14%
typeset	0.62	-0.12%	0.20	4.99%	0.52	2.16%
gsm.decode	0.43	0.01%	0.22	0.05%	0.25	-0.17%

Although all caches benefit from the static power reduction offered by drowsy caching, this technique has the most profound effect on the split heap caches. Drowsy caching all but eliminates the leakage energy of the large heap cache, as it contains rarely accessed data with low locality and is therefore usually inactive. Since the small cache experiences fewer conflicts in the split heap scheme than by itself, its lines are also less active and therefore more conducive to drowsy caching. Both techniques are very effective at reducing the energy consumption of these benchmarks.

Drowsy split heap caches save up to 76% of the total energy, while the small caches alone save between 77% and 84%. Because drowsy caching has a minimal performance cost, the runtime numbers are similar to those shown in the previous table. The small cache alone and the split heap cache produce comparable energy-delay values for several applications; *ispell* is one example. In these cases, performance-

conscious users can employ a split heap cache, while users desiring lower energy consumption can choose the small unified heap cache.

Shrinking the large heap cache further alleviates its effect on energy consumption. The data remaining in that cache is infrequently accessed and can therefore tolerate an increased number of conflicts. Table 8 shows simulation results for two different split heap configurations—one using a 32KB cache for cold heap data, the other using an 8KB cache—as well as the 2KB unified heap cache. All caches are drowsy. The unified cache is still most efficient for the majority of applications, but shrinking the cold heap cache narrows the gap between unified and split heap configurations. Applications such as *jpeg.decode* and *tiff2rgba*, which contain a non-trivial number of accesses to the cold heap cache, see the greatest benefit from this modification, with *tiff2rgba* consuming 37% less energy with the smaller cold heap cache. Overall, these applications save between 69% and 78% of the total energy.

Table 8. Energy and performance results for different drowsy heap caching configurations: a small heap cache alone, and split heap caches using 32KB and 8KB caches for low-locality heap data. The baseline is a 32KB direct-mapped unified L1 data cache with a 512 cycle drowsy interval. Shaded cells indicate the most effective heap caching method for each application.

Benchmark	2KB heap cache		Split heap cache (2KB hot/32KB cold)		Split heap cache (2KB hot/8KB cold)	
	Normalized total energy	Execution time increase	Normalized total energy	Execution time increase	Normalized total energy	Execution time increase
adpcm.encode	0.21	0.79%	0.26	0.45%	0.22	0.45%
adpcm.decode	0.21	0.65%	0.25	0.48%	0.22	0.48%
basicmath	0.23	1.20%	0.26	1.11%	0.24	1.11%
blowfish.decode	0.23	0.45%	0.24	0.20%	0.23	0.20%
blowfish.encode	0.23	0.48%	0.24	0.23%	0.23	0.23%
bitcount	0.23	0.00%	0.27	-0.04%	0.24	-0.04%
jpeg.encode	0.21	4.83%	0.29	3.59%	0.24	3.63%
CRC32	0.22	-0.36%	0.24	-0.73%	0.22	-0.73%
dijkstra	0.21	6.91%	0.24	5.99%	0.22	5.99%
jpeg.decode	0.19	11.70%	0.31	9.76%	0.22	9.99%
FFT	0.23	31.99%	0.27	7.35%	0.25	11.79%
FFT.inverse	0.23	22.28%	0.27	5.11%	0.24	8.20%
ghostscript	0.22	0.22%	0.26	-0.40%	0.24	-0.35%
ispell	0.23	2.84%	0.25	0.19%	0.24	0.68%
mad	0.21	0.47%	0.26	0.09%	0.22	0.10%
patricia	0.23	0.78%	0.27	0.62%	0.24	0.63%
pgp.encode	0.23	0.04%	0.26	0.00%	0.24	0.00%
pgp.decode	0.23	0.02%	0.26	-0.03%	0.24	-0.03%
quicksort	0.23	21.38%	0.29	22.48%	0.25	21.73%
rijndael.decode	0.22	1.99%	0.24	1.81%	0.22	1.81%
rijndael.encode	0.22	1.50%	0.24	1.29%	0.23	1.29%
rsynth	0.22	1.13%	0.25	0.65%	0.22	0.64%
stringsearch	0.23	0.22%	0.26	0.20%	0.24	0.19%
sha	0.23	0.14%	0.26	0.14%	0.24	0.14%
susan.corners	0.22	250.32%	0.45	-1.96%	0.27	48.34%
susan.edges	0.20	115.61%	0.48	-0.67%	0.27	22.48%
susan.smoothing	0.20	0.76%	0.33	-0.39%	0.24	-0.38%
tiff2bw	0.16	2.78%	0.55	1.15%	0.27	1.92%
tiffdither	0.19	7.44%	0.39	2.43%	0.25	2.57%
tiffmedian	0.17	4.99%	0.50	3.36%	0.26	4.01%
tiff2rgba	0.16	32.04%	0.68	0.05%	0.31	3.61%
gsm.encode	0.22	0.05%	0.24	-0.14%	0.23	-0.14%
typeset	0.20	4.99%	0.52	2.16%	0.29	3.30%
gsm.decode	0.22	0.05%	0.25	-0.17%	0.23	-0.17%

5 Conclusions

In this paper, we have evaluated a new multilateral cache organization designed to tailor cache resources to the individual reference characteristics of an application. We examined the characteristics of heap data for a broad suite of embedded applications, showing that the heap data cache footprint vary widely. To ensure that all applications perform well, we maintain two heap caches: a small, low-energy cache for frequently accessed heap data, and a larger structure for low-locality data. In the majority of embedded applications we studied, the heap footprint is small and the data possesses good locality characteristics. We can save energy in these applications by disabling the larger cache and routing data to the smaller cache, thus reducing both dynamic energy per access and static energy. This modification incurs a minimal performance penalty while reducing energy consumption by up to 79%. Those applications that do have a large heap footprint can use both heap caches, routing a frequently-accessed subset of the data to the smaller structure. Because a small number of addresses account for the majority of heap accesses, we can still reduce energy with both heap caches active—using up to 45% less energy—while maintaining high performance across all applications. When we implement drowsy caching on top of our split heap caching scheme, we can achieve even greater savings. With drowsy heap caches, disabling the larger structure allows for energy reductions between 77% and 84%, while activating both heap caches at once allows us to save up to 78% of the total energy.

In the future, we plan to further explore a few different aspects of this problem. We believe there is room for improvement in the heuristic used to determine which data belongs in which cache; we intend to refine that process. Also, the studies we ran using Cheetah suggest we can significantly lower the heap cache miss rate by reducing conflicts within it. We plan to investigate data placement methods as a means of ensuring fewer conflicts and better performance.

References

1. J. Montanaro, et al. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. *Digital Technical Journal*, (1):49-62, January 1997.
2. M.J. Geiger, S.A. McKee, and G.S. Tyson. Drowsy Region-Based Caches: Minimizing Both Dynamic and Static Power Dissipation. *Proc. ACM International Conference on Computing Frontiers*, pp. 378-384, May 2005.
3. H.S. Lee and G.S. Tyson. Region-Based Caching: An Energy-Delay Efficient Memory Architecture for Embedded Processors. *Proc. ACM/IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 120-127, November 2000.
4. H.S. Lee. Improving Energy and Performance of Data Cache Architectures by Exploiting Memory Reference Characteristics. Doctoral thesis, The University of Michigan, 2001.
5. K. Flautner, N.S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. *Proc. 29th IEEE/ACM International Symposium on Computer Architecture,* pp. 147-157, May 2002.
6. N.S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy Instruction Caches: Leakage Power Reduction using Dynamic Voltage Scaling and Cache Sub-bank Prediction. *35th IEEE/ACM International Symposium on Microarchitecure*, pp. 219-230, November 2002.

7. N.S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power. *IEEE Transactions on VLSI*, 12(2):167-184, February 2004.
8. M.R. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *Proc. 4th IEEE Workshop on Workload Characterization*, pp. 3-14, December 2001.
9. K. Ghose and M.B. Kamble. Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers and Bit-Line Segmentation. *Proc. ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 70-75, August 1999.
10. C-L. Su and A.M. Despain. Cache Designs for Energy Efficiency. *Proc. 28th Hawaii International Conference on System Sciences*, pp. 306-315, January 1995.
11. J. Kin, M. Gupta, and W.H. Mangione-Smith. Filtering Memory References to Increase Energy Efficiency. *IEEE Transactions on Computers*, 49(1):1-15, January 2000.
12. D.H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. *32nd IEEE/ACM International Symposium on Microarchitecture*, pp. 248–259, November 1999.
13. S.-H. Yang, M. Powell, B. Falsafi, and T.N. Vijaykumar. Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Eelay. *Proc. 8th International Symposium on High-Performance Computer Architecture,* pp. 147-158, February 2002.
14. S.-H. Yang, M.D. Powell, B. Falsafi, K. Roy, and T.N. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. *Proc. 7th International Symposium on High-Performance Computer Architecture*, pp. 147-158, Jan. 2001.
15. R.A. Sugumar and S.G. Abraham. Efficient Simulation of Multiple Cache Configurations using Binomial Trees. Technical Report CSE-TR-111-91, CSE Division, University of Michigan, 1991.
16. L.A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78-101, 1966.
17. T. Austin. SimpleScalar 4.0 Release Note. http://www.simplescalar.com/.
18. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *Proc. 27th IEEE/ACM International Symposium on Computer Architecture*, pp. 83-94, June 2000.
19. Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. Technical Report CS-2003-05, University of Virginia Department of Computer Science, March 2003.
20. D. Parikh, Y. Zhang, K. Sankaranarayanan, K. Skadron, and M. Stan. Comparison of State-Preserving vs. Non-State-Preserving Leakage Control in Caches. *Proc. 2nd Workshop on Duplicating, Deconstructing, and Debunking*, pp. 14-24, June 2003.
21. R. Gonzales and M. Horowitz. Energy Dissipation In General Purpose Microprocessors. *IEEE Journal of Solid State Circuits,* 31(9):1277-1284, September 1996.

Streaming Sparse Matrix Compression/Decompression

David Moloney, Dermot Geraghty, Colm McSweeney, and Ciaran McElroy

Department Of Mechanical & Manufacturing Engineering,
Trinity College Dublin, Dublin 2, Ireland
{moloned, tgerghty, mcsweece, Ciaran.mcelroy}@tcd.ie

Abstract. A streaming floating-point sparse-matrix compression which forms a key element of an accelerator for finite-element and other linear algebra applications is described. The proposed architecture seeks to accelerate the key performance-limiting Sparse Matrix-Vector Multiplication (SMVM) operation at the heart of finite-element applications through a combination of a dedicated datapath optimized for these applications with a streaming data-compression and decompression unit which increases the effective memory bandwidth seen by the datapath. The proposed format uses variable length entries which contain an opcode and optionally an address and/or non-zero entry. System simulations performed using a cycle-accurate C++ architectural model and a database of over 400 large symmetric and unsymmetric matrices containing up to 20M non-zero elements (and a total of 226M non-zeroes) demonstrate that a 20% average effective memory bandwidth performance improvement can be achieved using the proposed architecture compared with published work, for a modest increase in hardware resources.

1 Introduction

Many scientific and engineering problems require the solution of linear systems of equations of the form $A * x = y$, where A is the coefficient matrix of the system and x is a vector of unknowns and y is a vector of scalar known values. In practice A is large and sparse for real-world problems. A range of iterative methods is used depending on the nature of the problem to be solved and multiple solvers are normally included in commercial and public domain applications based on these methods.

Most implementations of iterative methods [1] and the linear algebraic operations on which they are based have to date been software implementations commonly implemented in the form of FORTRAN, C or C++ mathematical libraries which make use of the IEEE-754 [2] compliant floating point units (FPUs) included as part of high performance computing systems, microprocessors and workstations. Examples of such libraries include BLAS, ATLAS, MTL/ITL [3] and SPARSITY. All mathematical libraries in common use make use of data-structures to store sparse matrices efficiently and thus can be regarded as a form of compression.

The focus of this paper is on scientific and engineering applications primarily because a large body of test data is available for such problems [4][5][6]. However, linear algebra is emerging as enabling functionality in a range of applications such as

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 116 – 129, 2005.

spam-filtering [7] and face-recognition [8] which are embedded into infrastructural products today and may well find their way into more deeply embedded and even mobile devices in the future.

2 Related Work

Published works relevant to this work can be subdivided into four categories:

- Limitations of Existing Architectures
- Sparse Matrix Compression
- Trivial Arithmetic
- Special Purpose Computers (SPCs)

2.1 Limitations of Existing Architectures

The trend in processor micro-architectures has changed in the past 2-3 years, due to the difficulties in scaling deep sub-micron technologies, and currently all of the leading microprocessor manufacturers are moving away from technology scaling as the sole means of increasing performance, and towards new parallel architectures as a means of sustaining performance increases. In any case extensive work in the area of iterative solvers [9] for a range of applications on modern microprocessors has demonstrated that such architectures deliver less than 10% and often less than 1% of the quoted peak performance on such applications. The reason for this is the poor value reuse in the SMVM operation, where in practice for large real-world problems only the vectors containing the initial guess and result benefit from caching with the result that the processor spends most of its time waiting for 96-bit or 128-bit sparse-matrix entries (depending on storage format) to be fetched from external SDRAM over a 32-bit bus.

In fact the widening gap [10] between available memory bandwidth and processor performance seems to be relatively immune to process scaling and DRAM architectures have not evolved at a rate which keeps pace with CPU performance owing to in part to the commodity status of the DRAM business. In order to improve the effective memory bandwidth seen by any processor in a given process-technology interfacing to a common memory technology it was decided to investigate what, if anything, could be done to transparently implement streaming sparse-matrix compression on the way from memory to the processor and vice versa. In the case of the SMVM (Sparse-Matrix Vector Multiplication) in the context of an iterative solver the compression can be performed once and decompression may occur multiple times as most iterative methods take time to converge (require multiple SMVM operations) to solve non-trivial systems of equations.

2.2 Sparse Matrix Compression

A good overview of sparse-matrix storage methods employed in software libraries is given in [11]. These schemes typically consist of storing only the non-zero matrix entries along with the coordinates of those entries. In order to access the contents of the sparse matrix entry the row/column address reference must first be fetched and

this reference is then used to read the data value from a dense array of data values. All of the schemes in common use are variants of either row (compressed Row Storage CRS) or column major (Compressed Column Storage CCS) storage and assume that the data-structure needs to be fully random access. In CRS format each value is stored in a list structure together with its column index, and an additional structure containing pointers to the start of rows is also maintained. In this type of structure any column within a given row can be accessed efficiently allowing rapid traversal of the matrix on a row-wise basis. The same principle holds for CCS except that the matrix is traversed on a column-wise basis. All of these schemes operate at a word-level which is easy to map to a processor datapath and allows full random access.

In terms of recent work relevant to the topic of sparse-matrix compression a scheme outlined by Koster in [12] and called Incremental CCS (ICCS), makes use of relative rather than absolute row/column references is 30% faster than CCS (fewer instructions in critical loop). A number of other schemes have been proposed in [13] which make use of other arithmetic coding techniques such as run-length encoding in order to achieve additional compression. A similar technique has been applied to the lossless compression of single-precision floating-point meshes in computer graphics application as outlined by Isenburg in [14]. However, to date, no such scheme appears to have been implemented in hardware although work by Isenburg et al. continues on the implementation of a streaming version of their compression scheme aimed at reducing memory requirements and increasing compression speed.

2.3 Trivial Arithmetic

According to Richardson in [15] a significant number of trivial computations are performed during the execution of processor benchmarks and other numerically intensive applications. By trivial computations Richardson intended those that can be simplified or where the result is zero, one, or equal to one of the input operands. It was shown that for certain programs studied, up to 67% of operations were trivial and fast detection and evaluation of these trivial operations using dedicated hardware in the pipeline yielded significant speedup. However, to date, work on the exploitation of trivial operands appears to be focused on dynamically occurring trivial operands in the pipeline rather than static trivial operands occurring in the input data and no published work appears to detail the exploitation of trivial operands in sparse-matrix compression. Furthermore as the detection of and bypass of dynamically occurring trivial operands introduces problems of its own (ex. Pipeline bubbles due to the difference in FPU latency between trivial and non-trivial paths) such features have, to date, not appeared in commercial architectures.

2.4 Special Purpose Computers (SPC)

In the case of special purpose computer architectures such as White Dwarf [16], and SPAR [17] and more recently ILLIAD [18] customised sparse-matrix formats have been defined to which the architectures have been tuned in order to provide an overall increase in performance compared with software libraries running on general purpose processors (GPPs). SPC architectures have failed to become established in the face of the huge strides made in the design and manufacture of general purpose processors

from the mid-1980s with the introduction of RISC architectures and advanced process technologies.

3 Analysis of Sparse Matrix Data and Addresses

Relatively little has been published in the area of benchmarks for SMVM hardware architectures with the exception of [19]. In this work an effort was undertaken to construct a more extensive superset of the D-SAB database including very large problems. In order to evaluate potential compression and decompression strategies a database of over 400 symmetric and unsymmetric sparse coordinate matrices drawn from the Matrix-Market [5] and University of Florida (UF) matrix-collection [4] was profiled in order to determine the relative Row/Column offsets between matrix entries, and the number of trivial (compressible) non-zero matrix entries. And for the purpose of this work the definition of a trivial operand was expanded to include the following cases:

- trivial (+/-1) sparse-matrix entries
- +/-1e2^n sparse-matrix entries
- normalised sparse-matrix mantissae containing trailing zeroes

Initially the potential for non-zero data compression was evaluated for the complete matrix. It was found that out of the over 400 sparse matrices simulated, 129 show potential for compression of over 5% by compressing non-zero data values alone. An un-weighted average compression of 7.9% can be achieved for full matrix set as shown in Table 1.

Table 1. Data-only Compression

	+/-1 %	exp %	5-bit %	10-bit %	21-bit %	52-bit %	compr %	triv %
min	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
typ	***5.1%***	***2.0%***	***0.0%***	***0.0%***	***5.5%***	***85.2%***	***7.9%***	***12.6%***
max	86.0%	92.2%	0.0%	0.0%	100.0%	100.0%	66.7%	100.0%

It was also found that in the current matrix set none of the non-zero data values which contain a mantissa (ie not powers of 2) had less than 11-bits. Having established that the potential for compression based on the non-zero entries alone it was decided that the potential for compressing row and column addresses be assessed separately. Again by profiling the complete matrix database it was found that the average offset between successive columns in the set was at least 1 and a maximum of 4 which would allow column address displacements to be significantly reduced from the 32-bits required by conventional schemes.

As might be expected the variation in row-address displacements proved to be much larger than that in column addresses. In the complete set of over 400 matrices it was found that the maximum address range required for row addresses could be represented by 19-bits. In fact with the exception of but a few of the 100 worst-case matrices it can be seen that a row address range of 15 bits would allow all addresses to be represented as absolute addresses. Surprisingly no particular correlation was

found between row- address displacement and matrix size. In summary it was found that 19-bit relative addressing covers 100% of non-zero row entries in a column for entire matrix set, while14-bit relative addressing would cover 60% (worst-case) of non-zero entries for entire set. This represents 30% saving in address memory/bandwidth. In terms of column Addressing the worst case offset between successive columns is 4 requiring 2-bits, but the memory saving is modest because there are only N columns.

Finally the same collection was profiled to assess the potential for combined data/address compression. It was found that there is a large additional gain in terms of the percentage compression achievable by compressing row and column addresses in addition to non-zero data. Furthermore the potential for address compression is common to all matrices from the 400+ matrix set and not just the 25-30% which have potential for lossless data-compression.

4 Proposed Architecture

As shown in 16, compact formats such as CCS and RCS introduce three distinct types of overhead in accessing stored sparse-matrix data.

- High memory bandwidth
- Indirect accesses
- Short vector lengths

The SPAR format is a loose derivative of the CMNS format which introduces zero entries into the array of non-zero entries, which denote the end of each column. It is also augmented by the addition of a next-column entry to the address structure. The resultant SPAR data-structures are shown in Fig.1.

$$\vec{k}_v^T \left[k_{11}\ k_{31}\ 0.0\ k_{22}\ k_{52}\ 0.0\ k_{13}\ k_{33}\ k_{53}\ 0.0\ k_{44}\ 0.0\ k_{25}\ k_{35}\ k_{55}\right]$$
$$\vec{R}^T \left[1\,3\,2\,2\,5\,\mathbf{3}\,1\,3\,5\,4\,4\,5\,2\,3\,5\right]$$

Fig. 1. SPAR Data-Structure

The matrix-vector multiplication routine which operates on the SPAR data-structures is given below:

```
for (index=1; index <N_nz+N-1; index++) {
  if (K_v[index]==0.0) column=R[index];
  else v[R[index]] = v[R[index]] +K_v[index]*p[column];
}
```

For this reason it was decided to base the proposed storage format and compression scheme as well as the data-path on the SPAR architecture. The main difference between the proposed format and SPAR is that SPAR uses fixed-length 96-bit entries whereas the proposed format used variable length entries which contain an opcode

and optionally an address and/or non-zero entry. In terms of the streaming compression the objective was to improve over the SPAR scheme in terms of the storage and memory requirements, and more importantly in terms of the sustained performance which is limited in part by memory bandwidth in accessing the sparse-matrix data. The name given to the modified SPAR architecture is FIAMMA or Finite Iterative Arithmetic Matrix Mathematics Accelerator. The requirements for the design of the compression scheme were as follows:

- Maximise compression while minimising overhead (coding efficiency)
- Minimise encoder complexity, and maximise speed
- Minimise decoder complexity, and maximise speed
- Minimise re-ordering (numerical matching to software/hardware reference)
- Simple to implement in both hardware and software
- Single-pass algorithm suitable for streaming operation (minimal storage)

In order to be able to incorporate control information to implement the compression scheme along with the sparse matrix data it was decided that 5-bits from the 32-bit address-word defined in SPAR be used as an opcode to control the decompression of the compressed data. The 5-bit opcode format chosen is shown in Table 2.

Table 2. FIAMMA Opcode Format

	5-bit opcode					
Opcode	M	AL		ML		Matrix-Entry
VAM	0	x	x	x	x	addr/sign+exp+mantissa
TRU	1	x	x	0	0	addr/trivial +/-1
TRE	1	x	x	0	1	addr/trivial sign+exp
CLU	1	x	x	1	0	end-of-column marker
RES	1	x	x	1	1	reserved opcodes

As can be seen four opcodes and 4 reserved opcodes are encoded into 5-bits. The Address-Length (AL) and Mantissa Length (ML) fields within the 5-bit opcode tell the decoder the length of the encoded address and data so they can be efficiently decoded to reconstruct the uncompressed 27-bit address and 64-bit double-precision non-zero matrix entries without incurring any loss in numerical precision. The Address Length (AL) encoding is shown in Table 3 and allows the 2 bits to encode for 4 discrete lengths of address which can be chosen by the encoder in such a manner as to maximise address compression while ensuring no address information is lost. Along similar lines the Mantissa-Length (ML) 2-bit field encodes the 4 lengths shown in Table 4.

Table 3. Address-Length (AL) Encoding

AL	Length
00	27
01	19
10	11
11	3

Table 4. Mantissa-Length (ML) Encoding

ML	Length
00	13
01	26
10	39
11	52

The important feature of this encoding scheme is that the maximum length of an encoded address/data packet is 96-bits, and this combined with the length of the external memory subsystem word of 96-bits allows the design of the decoder to be simplified in that it can be guaranteed to be able to extract a single address/data pair as long as it has two 96-bit encoded words within it's internal memory. It will be shown later that while these numbers can be fixed there is an advantage to tuning them to a particular data-set in order to maximise the compression achieved. Also the 96-bit format allows backwards compatibility with the SPAR storage format allowing random access to matrix data stored in SPAR format using the same hardware and appropriate control logic.

Following extensive experimentation attempting to combine shortened mantissae and addresses within the limits of a 96-bit word it was decided that in order to maximise the potential for compression, compressed words should be allowed to straddle the boundary between 2 contiguous 96-bit words stored in external memory. This refinement complicates the compression and decompression logic but allows for much higher compression to be achieved. Importantly as the address/non-zero pairs are compressed and inserted in the data-structure in the same order as they arrive the compression scheme does not have the disadvantage of reordering non-zero entries which can lead to different results compared to software reference platforms owing to the accumulated effects of rounding in the IEEE-754 compliant FPU. This feature is very desirable in that it simplifies comparisons with reference software implementations, in that the rounding behaviour is identical to a software SMVM operation.

Finally, in the SPAR architecture the lack of support for symmetric matrix storage means that memory storage and memory bandwidth requirements are doubled compared to a symmetric storage scheme and the floating-point performance of a SPAR system without support for symmetric storage will be roughly half that of a an architecture which does have native support for sparse matrices. Essentially an architecture which stores symmetric matrices non-zero entries only once achieves almost 50% data/address compression. As can be seen in Table 2 in the case of +/-1 values with short row-address displacements the compression achieved can be very high where one such compressed entry occupies 9/96 of the memory required in the SPAR scheme. The additional benefit of compression is that more matrix-data can be fetched in a single cycle from external memory than would be the case in an uncompressed system. In the case of trivial +/-1s this could approach 10.67 (96/9) times the effective memory bandwidth of an unmodified SPAR architecture, assuming matrix compression and decompression does not become the system bottleneck. Similarly column updates which require 96-bits in a SPAR system require 9-bits in a FIAMMA system as the CLU opcode rather than the detection of a 64-bit 0.0 value denotes the end of a matrix-column.

As the proposed scheme does not alter the order of non-zero elements in the A-matrix and their associated row indices the compression has no degenerative effect on the performance of SPAR and the increase in matrix effective memory bandwidth translates directly into increased effective FPU utilisation.

4.1 Data Compression Implementation

Matrix compression is performed in a streaming manner on the matrix data as it is downloaded to the accelerator in a single pass rather than requiring large amounts of buffer memory allowing for a low cost implementation with minimal local storage and complexity. Where as in principle the compression can be implemented in software, in practice this may become a performance bottle-neck given the reliance of the compression scheme on the manipulation of 96-bit integers which are ill-suited to a microprocessor with a 32-bit data-path and result in rather slow software compression. The matrix compression logic consists of the following distinct parts:

- Delta-Address Calculation
- Address Compression
- Data Masking
- Compressed Entry Insertion (Write)
- Compressed Entry Retrieval (Read)

In current SMVM units whether implemented in hardware or software addresses are stored as absolute addresses; this is wasteful in terms of storage when based on the storage requirements of a large database of sparse matrices. The savings from such a scheme are significant and are easily implemented in hardware, both in terms of conversion from absolute to relative addresses and vice versa. The Delta-Address Compression logic keeps a record of the row and column base addresses as row or column input addresses are written to the block. These base addresses are then subtracted from the input address to produce an output delta-address under the control of the col_end input so the correct delta-value is produced in each case. Subtracting addresses in this manner ensures that the minimum memory possible is used to store address information corresponding to row or column entries as only those bits which change between successive entries need be stored rather than the complete address. The first stage in the compression of the address/non-zero sparse-matrix entries is to compress the address portion of the entry. The scheme employed is to determine the length of the delta-address computed previously so that the address portion of the compressed entry can be left-shifted to remove leading zeroes. Given the trade-off between encoding overhead and compression efficiency following extensive simulation it was decided that rather than allowing any 0-26-bit shift of the delta-address the shifts would be limited to one of four possible shifts. This both limits the hardware complexity of the encoder but also results in a higher compression factor being achieved overall for the matrix database used to benchmark the architecture. Before a shift to remove redundant leading bits in the delta-address can be performed the length of the quantised shift required must first be computed as shown below

```
addr_bits = log_2(a_ij.addr);
if (addr_bits>19) addr_comp = 3;
else if (addr_bits>11) addr_comp = 2;
else if (addr_bits> 3) addr_comp = 1;
else  addr_comp = 0;
q_addr_bits = al_len[addr_comp];
v_addr = (a_ij.addr & ((1<<q_addr_bits)-1));
```

In line 1 leading-one detection is performed and rounded up to the next highest power of 2 to allow for the trailing bits in the address (achieved by adding an offset of 1 to the position of the leading one). An efficient leading-one detector is described in [20]. The addr_bits signal generated by the LOD is then compared using 3 magnitude comparators to identify the shift range required to remove leading ones, and finally the outputs of the comparators are combined as shown in Table 5 to produce a 2-bit code.

Table 5. Delta-Address Encoder Truth-Table

			addr_comp	
addr>19	addr>11	addr>3	[1]	[0]
1	0	0	1	1
0	1	0	1	0
0	0	1	0	1
0	0	0	0	0

One key advantage of the address-range quantisation scheme is that the shifter consists of 4 multiplexers implementing 3 simple hardwired shifts rather than a complex bit-programmable shifter with many more multiplexers which would be required if any shift in the range 0-26 bits were used in the compression scheme. Once an address/data entry has been compressed into a shortened format must be inserted into the FIAMMA data-structure in memory. The FIAMMA data-structure is a linear array of 96-bit memory entries and in order to achieve maximum compression each entry must be shifted so it is stored in a bit-aligned manner leaving no unused space between it and the previous compressed entry stored in the FIAMMA array. There are three cases which can occur in inserting a compressed address/non-zero word into a 96-bit word within the FIAMMA data-structure in memory:

- The compressed entry when inserted leaves space for a following entry within the current 96-bit FIAMMA entry
- The compressed entry when inserted fills all of the available bits within the current 96-bit word
- The available bits in the current FIAMMA 96-bit memory word are not sufficient to hold the compressed entry and part of the compressed entry will have to straddle into the next 96-bit FIAMMA memory location

One point worth noting is that the compressed entry insertion mechanism is independent of the actual compression method utilised and hence other compression schemes such as those detailed in [13] could in principle be implemented using the unmodified FIAMMA data-storage structure as long as the compressed address/data entries fit within the 96-bit maximum length restriction for compressed FIAMMA entries.

4.2 Data Decompression Implementation

As in the case of the data-compression path, decompression is performed in a streaming manner on the compressed packets as they are read in 96-bit sections from external memory. Performing decompression in a streaming fashion allows decompression to be performed in a single pass without having to first read data into a large buffer memory. The complete data and control path for data decompression is shown in Fig. 2. As can be seen the decompression path consists of control logic which advances the memory read pointer (entry_ptr) and issues read commands to cause the next 96-bit word to be read from external memory into the packet-windowing logic.

This is followed by address and data alignment shifters, the address shifter correctly extracts the delta-address and the data alignment shifter correctly aligns the sign, exponent and mantissa for subsequent data masking and selection under

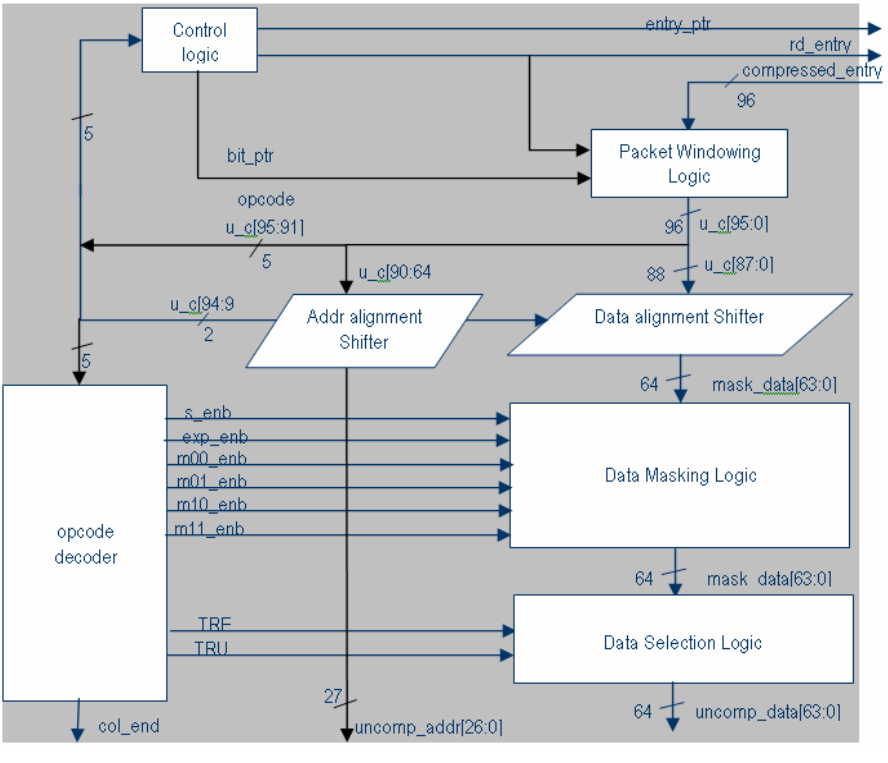


Fig. 2. FIAMMA Decompression Path

the control of the opcode decoder. In order to ensure that the opcode can be properly decoded in all cases a 192-bit window must be maintained which straddles the boundary between the present 96-bit packet being decoded and the next packet so the opcode can always be decoded even if it straddles the 96-bit boundary. The windowing mechanism is crucial to the proper functioning of the decompression logic as the opcode contains all of the information required to correctly extract the address and data from the compressed packet. The pseudocode for the packet-windowing logic is shown below:

```
available = (96 - bit_ptr);
if (available<96) u_c = (entries[entry_ptr]<<(96-
available))|(entries[entry_ptr+1]>>available);
else u_c = entries[entry_ptr];
```

The decompression logic works by moving a 96-bit window over the compressed data in the fiamma data-structure as the maximum opcode/addr/data packet length is always 96-bits in the compressed format so the next 96 bits is always guaranteed to contain a compressed fiamma packet. The design consists of a comparator which detects if the codeword straddles two 96-bit compressed words in memory. In the event a straddle is detected a new data word is read from memory from the location pointed to by entry_ptr+1and the data-window is advanced, otherwise the current data-window around entry_ptr is maintained in the two 96-bit registers. The contents of the two 96-bit registers are then concatenated into a single 192-bit word which is shifted by bit positions to the left in order that the opcode resides in the upper 5 bits of the extracted 96-bit field so the decompression process can begin. In order for decompression to proceed correctly it must adjust the entry_ptr pointer which points to the current 96-bit compressed word being operated on, and the bit_ptr pointer to the beginning of the next opcode within that word. In order to correctly adjust these pointers the length of the compressed word starting at location bit_ptr in the current compressed entry must be determined using the opcode field pointer to by bit_ptr. A simple look-up table is used to generate the len value used in the decompression control logic. The len value is then used to update the bit_ptr and entry_ptr values as shown below:

```
available = (96 - bit_ptr);
full = len + bit_ptr;
if (full > 96) {
  entry_ptr++;
  bit_ptr = (len - available);
}
else if (full == 96) {
  entry_ptr++;
  bit_ptr = 0;
}
else bit_ptr += len;
```

The address-field is decompressed by decoding the AL sub-field of the opcode which always resides in the upper 5 bits of u_c[95:0], the parallel shifter having performed a normalization shift to achieve this objective. Once the delta-address information has been correctly aligned it must converted back to an absolute address by adding the appropriate column or base address. A similar 96-bit shifter is used to correctly align the compressed data and this together with data selection and masking logic allows the original uncompressed non-zero data entries to be reconstructed in a lossless manner. The estimated hardware cost of the streaming decompression logic is detailed in Table 6 and comes to a total of around 4k gate equivalents assuming the 14 logic levels between alignment shifter, data masking logic and data selection logic can operate at the required clock-rate without pipelining. Assuming that the 192-bit shifter is the critical path, each level of pipelining if required will introduce approximately 123 (96+27) pipeline flip-flops to add to the base 232 flip-flops.

Table 6. Decompressor Gate-count

Sub-Block	*FA*	*Mux*	*Gates*	*FFs*
control logic	100	16	712	40
packet windowing	16	672	2176	192
addr alignment shifter		81	243	
data alignment shifter		192	576	
Data Masking Logic			74	
Opcode Decoder			10	
Data Selection Logic		128		
Total			3791	232

To put this in context, synthesis from SystemC RTL of the double-precision floating-point multiplier reported in [21] comes in at approximately 75k gates, so the overhead of the streaming decompression-logic comes to about 5% of that of an IEEE754 compliant double-precision multiplier. As can be seen from the table the dominant block in terms of both logic-levels (critical path) and gate-count is the 192-bit shifter. The implementation cost of the streaming compressor is of the same order as the Decompressor and as previously can also be implemented in software in the host.

5 Results and Conclusions

In order to validate the proposed architecture a cycle-accurate C++ model was constructed. Previous related work on the design of floating-point adders and multipliers in SystemC RTL detailed in [21] shows that no penalty is incurred in terms of hardware efficiency by implementing designs in SystemC compared to Verilog or VHDL. The model allows matrices to be stored in either unmodified SPAR format or FIAMMA format and to be compared to a reference MTL SMVM (Sparse Matrix-Vector Multiplication) implementation for the complete matrix set. The design is implemented as a set of C++ classes which allow Matrix-Market files to be read and report files to be generated which can be read by a spreadsheet package. The model also includes a range of

cache models allowing the architecture to be tuned for maximum performance by varying cache parameters such as line length and number of double precision entries per line. System simulations using a C++ architectural model using a database of over 400 large symmetric matrices demonstrate that an average 20% increase in effective memory bandwidth can be achieved for sparse-matrices stored in the proposed format when compared to the reference SPAR format. The same figure compares the ideal scheme with that actually implemented in the C++ model. Importantly the improvement in performance comes at a moderate incremental cost in terms of hardware which consists primarily of adders, shifters and decoding logic as well as a few registers to store base addresses used in the relative addressing scheme.

In order to improve the compression achieved by the implementation the distribution of delta-address lengths seen by statistical analysis of the matrix database showed many of the address displacements were very short, for instance column address displacements were on the order of a bit or two and the fact that even locally within rows data tends to be clustered. For this reason two alternate address-length range encodings corresponding to the opcode AL field were modeled as shown in Table 7.

Table 7. Opcode Address-Length (AL) Encoding

Coding Scheme	*11*	*10*	*01*	*00*
AL_enc_1	27	24	16	8
AL_enc_2	27	19	11	3

Simulation showed that the AL_enc_2 encoding scheme increased the average compression achieved across the entire matrix database by approximately 3%. The reason for this improvement in terms of average compression is the number of 3-bit displacements in the matrix database is quite high, and quantizing them to 8-bit displacements using the AL_enc_1 scheme is wasteful in terms of storage. The main difference between the proposed format and SPAR is that SPAR uses fixed-length 96-bit entries whereas the proposed format used variable length entries which contain an opcode and optionally an address and/or non-zero entry. Overall the scheme works by exploiting the fact that access to the matrix data-structure does not have to be random access and this fact is exploited in order to compress matrix data in the same order it will be accessed by the sparse matrix-vector multiplier thus leading to a significant decrease in storage requirements and memory bandwidth.

Work on exponents has shown that there is potential to transform over 89% of the exponents of non-zeroes from an 11-bit range to a 5 bit range by adding a constant offset as the bulk of the 226M non-zeroes in the 400+ matrices occur in a relatively narrow range although the scheme implemented here does not include exponent compression.

References

1. R. Barrett , M. Berry , T. F. Chan , J. Demmel , J. Donato , J. Dongarra , V. Eijkhout , R. Pozo , C. Romine and H. Van der Vorst, “Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, SIAM, 1994, Philadelphia, PA

2. IEEE Standards Board, "IEEE Standard for Binary Floating-Point Arithmetic", Technical Report ANSI/IEEE Std. 754-1985, IEEE, New York, 1985
3. J. Siek & A. Lumsdaine, "The Matrix Template Library: Generic Components for High-Performance Scientific Computing", IEEE Journal of Computing in Science & Engineering, Nov.-Dec. 1999, pp.70-78
4. http://www.cise.ufl.edu/research/sparse/
5. http://math.nist.gov/MatrixMarket/
6. http://www.parallab.uib.no/projects/parasol/data/
7. Kevin R. Gee, "Using latent semantic indexing to filter spam", SAC '03: Proceedings of the 2003 ACM symposium on Applied computing, pp.460-464
8. N. Muller, L. Magaia, B. M. Herbst, "Singular Value Decomposition, Eigenfaces, and 3D Reconstructions", SIAM Review, Vol. 46, No. 3, pp. 518-545
9. W. K. Anderson , W. D. Gropp **,** D. K. Kaushik, D. E. Keyes and B. F. Smith, "Achieving high sustained performance in an unstructured mesh CFD application"**,** Proceedings of the 1999 ACM/IEEE conference on Supercomputing, , No. 69, Portland, Oregon, United States
10. B. Jacob, "A case for studying DRAM issues at the system level." *IEEE Micro*, vol. 23, no. 4, pp. 44-56. July/August 2003
11. I. S. Duff, A. M. Erisman and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, London, 1986
12. J. Koster, "Parallel templates for numerical linear algebra, a high-performance computation library", MSc. Thesis, Dept. of Mathematics, Utrecht University, July 2002
13. Bell, T.; McKenzie, B.; "Compression of sparse matrices by arithmetic coding**",** Data Compression Conference, 1998. DCC '98. Proceedings, 30 March-1 April 1998 Page(s):23 – 32
14. M. Isenburg, P. Lindstrom, J. Snoeyink, "*Lossless Compression of Floating-Point Geometry"*, Proceedings of CAD'3D, May 2004
15. S. Richardson, "Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation", International Symposium on Computer Arithmetic, 1993.
16. A. Wofle, M. Breternitz, C. Stephens, A.L. Ting, D.B.Kirk, R.P.Bianchini, J.P.Shen "The White Dwarf: A High-Performance Application-Specific Processor"
17. Taylor V.E.; Ranade A.; Messerschmitt D.G, **"**SPAR: a new architecture for large finite element computations**"**, IEEE Trans. on Computers, Vol. 44 , No. 4, April 1995, pp.531 – 545
18. P.T. Stathis, S. Vassiliadis, S. D. Cotofana, A Hierarchical Sparse Matrix Storage Format for Vector Processors**,** Proceedings of IPDPS 2003, pp. 61a, Nice, France, April 2003
19. P.T. Stathis, S. Vassiliadis, S. D. Cotofana, D-SAB: A Sparse Matrix Benchmark Suite**,** Proceedings of 7th International Conference on Parallel Computing Technologies (PaCT 2003), pp. 549-554, Nizhni Novgorod, Russia, September 2003
20. M. Schmookler & K. Nowka. "Leading-Zero Anticipation and Detection – A Comparison of Methods", *Proceedings of IEEE Symposium on Computer Arithmetic, pp7-12*, 2001.
21. D. Moloney, D. Geraghty, F. Connor, "The Performance of IEEE floating-point operators on FPGAs", Proc. Irish Signals & Circuits Conf. (ISSC) 2004, pp. 254-259

XAMM: A High-Performance Automatic Memory Management System with Memory-Constrained Designs

Gansha Wu[1], Xin Zhou[1], Guei-Yuan Lueh[3], Jesse Z Fang[2], Peng Guo[1], Jinzhan Peng[1], and Victor Ying[1]

[1] Intel China Research Center, 100080 Beijing, China
{gansha.wu, xin.zhou, peng.guo, paul.peng, victor.ying}@intel.com
[2] Intel Coporation, Corporate Technology Group, 95052 CA, USA
jesse.z.fang@intel.com
[3] Intel Coporation, Software Solutions Group, 95052 CA, USA
guei-yuan.lueh@intel.com

Abstract. Automatic memory management has been prevalent on memory / computation constraint systems. Previous research has shown strong interest in small memory footprint, garbage collection (GC) pause time and energy consumption, while performance was left out of the spotlight. This fact inspired us to design memory management techniques delivering high performance, while still keeping space consumption and response time under control. XAMM is an attempt to answer such a quest. Driven by the design decisions above, XAMM implements a variety of novel techniques, including object model, heap management, allocation and GC mechanisms. XAMM also adopts techniques that can not only exploit the underlying system's capabilities, but can also assist the optimizations by other runtime components (e.g. code generator). This paper describes these techniques in details and reports our experiences in the implementation. We conclude that XAMM demonstrates the feasibility to achieve high performance without breaking memory constraints. We support our claims with evaluation results, for a spectrum of real-world programs and synthetic benchmarks. For example, the heap placement optimization can boost the system-wide performance by as much as 10%; the lazy and selective location bits management can reduce the execution time by as much as 14%, while reducing GC pause time on average by as much as 25%. The sum of these techniques improves the system-wide performance by as much as 56%.

1 Introduction

Java is increasingly prevalent on memory / computation constraint systems. Accordingly there appears strong interest in deploying automatic memory management on these systems. The design philosophy for these systems differs from that of desktops or servers. For example, desktop or server runtimes may tradeoff memory for performance. Nevertheless the top challenging task here is to

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 130–149, 2005.

design a system that always fits into a given space quota while, if possible, still having acceptable response time and performance. Researchers have extensively investigated in the aspects of memory footprint and GC pause time. Performance is usually not their No.1 consideration.

This paper presents XAMM, a high-performance memory management system optimized for platforms with Intel XScale^{TM} technology. We will describe design decisions and implementation trade-offs that balance performance, memory usage and response time. Our test-bed platform positions at the high-end spectrum of mobile devices, as such we may not consider the space limit extremely like [15]; meanwhile XAMM is not targeted for real-time systems (e.g. [7]), or rather it aims at applications that allow a worst-case response time of tens of micro-seconds for a heap with tens of MBs. Our research focus is to boost the performance as aggressive as possible, while still keeping memory consumption and response time controlled.

The contributions of this paper are in describing the following distinguishing aspects:

1. XAMM integrates a set of innovative but general techniques into one framework, and shows the feasibility to excel in all the following design spaces without compromising each other: high performance, small and predictable memory usage, short average pause time, and simplicity of implementation.
2. XAMM adapts common memory management practices to exploit platform-dependent features. For example, it modifies the classic mark-scan algorithm to apply prefetching techniques.
3. XAMM leverages the knowledge of memory management to assist efficient code generation. It expands the role of memory management to enable system-wide optimizations.

The remainder of the paper is organized as follows. Section 2 presents the high-level design of XAMM. Section 3 describes several innovative techniques in XAMM. Section 4 discusses experimental results in support of our designs. Section 5 briefly describes related work, and Section 6 concludes the paper by summarizing our findings.

2 XAMM Overview

XAMM is integrated with XORP, a clean-room runtime system designed specifically for high performance and small memory footprint. XORP has full-fledged support for two J2ME configurations, namely CLDC and CDC. Figure 1 presents the module view of XORP. Boxes with darker shadows are platform dependent; the light-shadowed parts are configuration dependent and software-switchable; and the rest parts are shared by all platforms and configurations. XORP VM acts as the hub that connects different components. The execution engine employs an adaptive compilation scheme that has threaded interpreter and a JIT run in parallel. Platform Abstract Layer conceals OS and architecture specifics (e.g. threading and signaling) and exposes common interfaces to VM. Machine

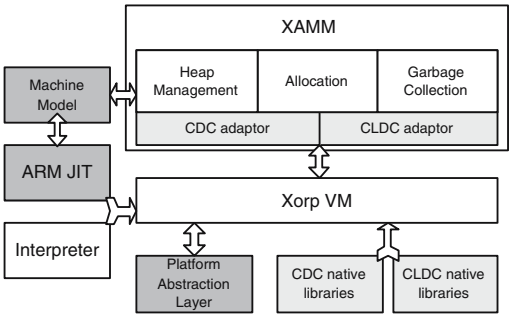


Fig. 1. XORP: the module view

Model instructs JIT and XAMM perform aggressive platform-related optimizations. XAMM consists of three modules, heap management, allocation and GC, and Section 2 has a high-level overview of these modules.

2.1 Heap Management

As shown in Figure 1, XAMM is largely platform independent, but it is still configurable by Machine Model and configurations. The VM-XAMM interface defines a compact object model, and the one-word header design (explained later) reduces space budget per object without sacrificing performance visibly.

The heap management employs an intelligent heap placement strategy, which enables better code generation, and a hierarchical heap layout to meet allocation requirements flexibly. The heap is composed of *chunks*, the allocation unit that XAMM obtains from underlying system's memory manager. Chunks are made up of *blocks*, the place where XAMM allocates objects. Several contiguous blocks may constitute a macro-block to accommodate large objects. When the heap is too fragmented to afford contiguous blocks, our discrete array design can still allocate large arrays into discrete blocks, while still accessing them efficiently.

2.2 Allocation

Segregated lists based allocation is widely used (e.g. [11][6]), for the sake of fast allocation and low fragmentation. We adopt a different scheme, block sequential allocation and sliding compaction, to achieve the same goal. Allocation within a block is sequential, and it can be as fast as simply bumping a free pointer (when the block is free of fragmentation). Sliding compaction removes fragmentation, and preserves locality by keeping objects' natural allocation order, which results in better memory behavior. Such a system, if deliberately designed, can outperform segregated lists based allocation significantly. A block may be affiliated with a specific thread between GC cycles. The affiliation enables lock-free allocation, which is a significant performance win in native threading environments. The affiliation must be broken and reconstructed for each GC, to guarantee fair distribution of blocks among threads.

The total runtime memory footprint consists of four parts: managed heap space, global and static data area, unmanaged heap space (e.g. space allocated by `malloc`), and unmanaged runtime stack. Since the excessive use of the last two may jeopardize the predictability of the runtime footprint, XAMM adopts unified memory management, called UMM for short, to allocate language objects[1] and vast majority of VM data structures in the managed heap. Only a few performance-sensitive temporary data and very short-lived small VM objects still inhabit in the unmanaged space. UMM not only collocates VM and language objects, but also arranges language objects and JIT'ed code side by side. The collocation yields better cache behavior and opens up more optimization opportunities (e.g. heap placement). UMM automatically reclaims unreachable VM objects, which alleviates the complexity of managing space and debugging memory issues. XAMM is smart enough to identify garbage from seemingly reasonable memory usages. For instance, VM maintains a global pool for interned strings. Those strings could be always treated as live by a naive reachability analysis starting from the pool. So are loaded classes and compiled code. XAMM employs a weak-reference based unloading mechanism to reclaim them safely[2]. In addition, UMM can execute tailored policies for different allocation requests, to name a few: allocate immortal objects in permanent blocks to save scavenging cost, or aggregate hot JIT'ed code to improve the efficiency of I-Cache / ITLB.

2.3 Garbage Collection

Adopting UMM comes with challenges as well. Many VM routines may trigger GC and it's cumbersome to spread managed pointers all over the VM code. We employ conservative GC[13] to avoid switching GC safe and unsafe states back and forth, and avoid maintaining handles to managed pointers. Conservative GC also eliminates the complexity to create a fully-cooperative environment, for example, the JIT compiler doesn't necessarily provide support for precise stack frame enumeration. In this regard, we compromise GC's conservatism for the simplicity of the VM implementation. But XAMM GC, or the so-called *Enhanced Conservative GC* (ECG), is sophisticatedly designed to be less conservative than conventional implementations. It's conservative only for stack scanning, while the heap traversal is accurate. At the beginning of a GC cycle, a stack scanning routine walks through every slot of call stack – indiscriminately for Java stack and native stack – and identifies slots that happen to contain legitimate reference values. Later in this paper, we will present several mechanisms, including lazy and selective location bits management and selective stack clearing, to screen out spurious pointers on the stack so as to reduce falsely retained dead objects.

[1] The term "language object", contrasted to "VM object", refers to an object allocated by Java programs.

[2] For class unloading, we ensure no side effects take place by tracking static variables, static initializers and native methods[14].

XAMM deploys a mark-sweep [19] GC with incremental compaction. Each block has a dedicated bitmap to serve as mark bits. Since the mark-scan routine generally accounts for the largest portion of total GC pause time, we revised the conventional process by enabling prefetching. Sweeping is another time-consuming process. GC must walk over mark bits to demarcate new free segments. These free segments must be cleared before they are ready to allocate. The lazy sweeping scheme delays the clearing operation until the segment is about to be allocated. In other words, the clearing overhead is spread over the *object allocation* phase. The object allocation routine pays penalty to check a *swept* flag for each allocation request, but GC pause time is effectively reduced. Lazy sweeping is not new, but we ameliorate it by performing the clearing task at *block allocation* phase. As aforementioned, all blocks are detached from their affiliated threads at the end of a GC cycle; after GC resumes the world, XAMM re-assigns blocks to threads on demand. When assigning a block to a new thread, all the free segments in the block are cleared at once. This design leads to considerable performance benefit for allocation-intensive applications.

The compaction algorithm is an optimized variant of break-table based compaction[18]. As a form of sliding compaction, it is inherently cache friendly. And it's more than a conventional sliding compaction. Firstly, its incremental property aims to compact only a small part of the heap to make the compaction time bounded. At the beginning of each GC cycle, XAMM selects compacting chunk candidates, based on some history and runtime heuristics, to make each compaction most profitable. Secondly, we use free heap space to accommodate the break table to avoid extra memory overhead, and layout table entries contiguously so as to eliminate the rolling and sorting time. Last but not the least, conventional break-table implementations (e.g. [16]) apply to a contiguous heap space, but our algorithm practices a block-by-block scheme. And moving objects is not restrained by block or chunk boundaries: live data can be compacted across chunks, which frees up more contiguous space.

3 Innovative Designs

3.1 Design of Compact Object Model

Modern object-oriented programming languages usually impose some space overhead on each language object, namely an object header, to support various runtime features. Object model is the representation of the header storage. The size and structure of the object header, which is transparent to users, are crucial to system-wide performance. State-of-the-art VM implementations demonstrate a great diversity of designs, varying from two words (e.g. ORP[2] and Jikes RVM[1]) to three words (e.g. IBM JDK 1.3.1[4]). A typical two-word header design may look like:

```
struct obj_info {
    class_info* ci;
    uint32 state;
};
```

Where `class_info` pointer pointer assists virtual method dispatching and runtime type test; `state` may be encoded to facilitate GC, synchronization, and instance identification (e.g. hash code). This design imposes high space pressure for systems with a small heap. Our experiment shows object headers may occupy about 9%∼36% of total allocated space, and even more for applications with large population of small objects. From this perspective, one-word header can alleviate the space pressure. But on the other hand, one-word header may lose some performance advantages that a full-fledged header provides, because it's hard to squeeze all necessary information into one word.

Bacon et al.[8] proposed that objects are born with one-word header, but switch to two-word header when they are involved in hashcode or lock operations. Monty[3] (the predecessor of CLDC-HI) also advocates a one-word design: the header contains a pointer to a so-called "near object", which in turn stores `class_info`. The near object may be redirected or inflated when lock or hashcode is requested. This design pays one more pointer chasing for each virtual method invocation. What's more, small systems may not afford the complexity to maintain two header layouts. A uniform one-word model may be more desirable for them. Indexed object model[8] can fit into the uniform design: Store an index into a `class_info` array instead of `class_info` pointer itself, and thus leave more bits for runtime state. This model leads to slower virtual method dispatching. The number of bits allocated for the index puts limit on the scale of applications, i.e. how many classes can be loaded.

Our compact object model has a different design philosophy: We encode the header word in such a way that it delivers best performance for the most common cases, and pays penalties only for infrequent cases. In practice, we obey the following principles: (1) Allow fast method dispatching; (2) Approximate the performance of thin lock[9]; (3) Generate effective hashcode for most small-scaled applications; (4) Provide extensibility if we request more bits for new usages.

The object model is based on a bit-stealing technique[8]. Basically the header word stores the `class_info` pointer. We also steal some bits from the word to encode more information. The address of `class_info` can be properly aligned to a power of 2 so that some least significant bits (LSBs) of the word are available for other usages. Compared with conventional bit-stealing techniques, our object model is more aggressive: we explicitly align `class_info` to a large value; consequently the header word can offer more tailing bits for information encoding.

Figure 2 depicts the layout of the object header:

- *`class_info`*: `class_info` is allocated on the boundary of 2^N. During virtual method dispatching, the `class_info` pointer can be obtained by clearing N LSBs of `this` object's header. It is nearly as efficient as two-word design since bit-clearing operations are very cheap for most modern architectures.
- *Reserved bits*: These bits are reserved for future usage. We can acquire more bits by aligning `class_info` to a larger value.
- *Hashcode bits*: We assign 3-4 bits as seeds for hashcode generation.
- *Lock bits*: 2 bits are dedicated for lock implementation.
- *Reflection bits*: We allocate one or more bits for fast reflection support, e.g. whether the object is a VM or language object, whether it is an array, etc.

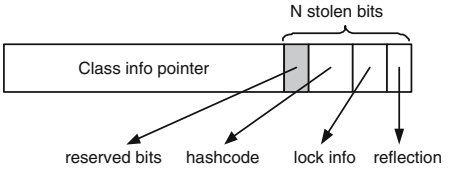


Fig. 2. Compact object model for language object

We design a lightweight lock by using two bits in the header. The rationale is that locks are most frequently performed by thread #1 (main thread) without recursive locking and contention, which we call an *easy lock*. For the two lock bits: Lock bit 1 is a flag indicating whether the object has been locked; lock bit 2 indicates whether the object is locked in a *easy* manner. The two bits set to 00 represents a free lock; 10 indicates the object is locked by an *easy lock*; when the lock is set to 11, the object can be either locked by thread n ($n \geq 2$), recursively locked by thread #1, or under contention. We say heavy locks are involved in these scenarios. The two-bit lock performs almost as well as thin-lock for most applications we studied, while thin-lock implementations typically require at least one word.

We use three or four bits as a hashcode seed. For typical containers on small systems, our hashcode generator can keep collisions under a reasonable threshold. Our workload favors this design: One reason is that besides the hashcode seed, the hash algorithm also takes `class_info` into consideration to reduce collision for heterogeneous containers (a container can have elements of various types); for many homogeneous containers, the elements may override the default hashcode generator by their own `hashCode()` implementation. For instance, a `java.lang.String` object generates a hashcode based on its characters.

Aligning `class_info` to 2^N comes with fragmentation. Given that `class_info` typically attributes less than 1% of total object population, the overhead of fragmentation is negligible.

3.2 Heap Placement

Intel XScale^{TM} core implements the ARM ISA. Due to the RISC nature of the ISA, there're certain constraints on expressing immediate values. For instance, `B` (branch) and `BL` (branch with link) require that the distance between current PC and the target must be smaller than 32M; `MOV` requires the immediate source operand to be an 8-bit value with 4-bit rotation. These constraints become a burden for optimized compilers. XAMM takes advantage of the hosting environment's memory map, or virtual address space layout, to perform a best-effort heap placement strategy that leads to optimal code sequences.

A typical OS has the memory map as shown in Figure 3:

- Code region, or the `text` segment, stays at the lower end of the address space.
- Static data region, namely the `data` segment for initialized data and the `bss` segment for uninitialized data, are adjacent to the code region.

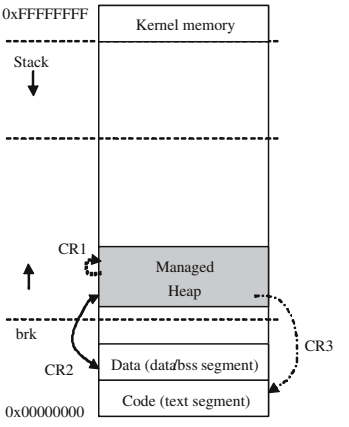


Fig. 3. The memory map after intelligent heap placement

– At the high end of the space, OS reserves a space dedicated for kernel use.
– The above three spaces are the "static" parts of the address space, and the in-between space, occupied by the stack and heap, is "dynamic".

Memory management libraries select some region(s) in the dynamic space for heap allocation. The location of the heap region depends on various design choices or runtime situations; typically, the heap inhabits at some addresses that are far away from the static data region. For ARM Linux, the distance between a dynamic heap and the static data region can be more than 1G bytes. Our strategy, however, is to push the heap as close to the static data region as possible so that the relative offsets are small to fit into the immediate values. In Figure 3, the label "brk" indicates the line of the boundary. To obtain a heap around this line safely, we can either move the line upward (e.g. via `sbrk`) and put the heap below the line, or reserve a space (e.g. via `mmap`) above the line. We call this technique *heap placement*.

XORP's JIT compiler adopts several techniques that magnifies the ISA's capability to express immediate values. The essence of these techniques is to deal with cross references, for example, cross references within the heap, or between the heap and unmanaged areas.

Figure 3 illustrates such cross references with directed edges:

1. CR1 represents cross references within the heap, e.g. a JIT'ed code calls another JIT'ed code; a JIT'ed code loads an object address; an object's field points to another object.
2. CR2 are the references from heap to static data, such as: heap objects may refer to static data, and a JIT'ed code loads an address in the static area.
3. CR3 refers to those references from dynamic code to static code. A typical example is that JIT'ed methods invoke VM helper routines.

One code generation rule is to generate direct branches from JIT'ed code to VM code (CR3). Consider an illustrative example: Suppose the code address for

the VM routine `checkcast` is 0x1cbc8, and the address for a JIT'ed code can be as high as 0x40000000 without lowering the heap. The distance between the JIT'ed code and `checkcast` goes far beyond the limit of 32M, and the JIT compiler may have to emit the code as follows[3]:

```
63 checkcast #25 <Class SomeClass>
401cf274        mov r12, 0xc8
401cf278        orr r12, r12, 0xcb00
401cf27c        orr r12, r12, 0x10000 % checkcast=0x1cbc8
401cf280        blx r12
```

Our heap placement strategy dramatically shortens the distance between call sites and targets. Resultingly the JIT compiler can emit the code as simple as:

```
63 checkcast #25 <Class SomeClass>
4cd3ac          bl 0x1cbc8    % checkcast=0x1cbc8
```

The instruction sequence is shorter. What's more, direct calls (`BL` in this case) are superior to indirect calls (`BLX` in the code above), because the latter is subject to serious micro-architectural penalties, such as branch mis-predication and pipeline flushing. Since JIT'ed code frequently calls back on VM helper routines to complete runtime operations (for example, allocation and type checking), we have substantial performance gain from this technique.

Another low hanging fruit is to construct object addresses (CR1) efficiently. JIT'ed code frequently refers to addresses of pinned objects, for instance, a known class or method handle. Conventional compilers tend to use a literal pool to load and store these addresses, which leads to more load stalls on the critical path. With a lowered heap, these values are usually small and regular. So the JIT compiler can construct them on the fly with lightweight instructions. For example, the code for loading a method handle (0x5f864) can be as simple as:

```
4cd3ac          mov r0, 0x5f800
4cd3b0          add r0, r0, 0x64 %handle=0x5f864
```

3.3 Discrete Array Allocation

Allocation of large objects is a problem for systems with limited memory. Chen et al.[15] try to compress objects when compacting the heap still can't free up enough space; the compressed objects will be decompressed upon accesses. Upon decompressing the objects, it still requires a contiguous full-sized space. This design potentially incurs a considerable performance cost.

Other research has addressed this problem by treating arrays specially. Bacon et al.[6] transform all arrays into arraylets. An arraylet manages multiple discrete parts but gives the illusion that all array elements reside in a virtual contiguous space. The runtime overhead of arraylets is not as high as [15], but is still high.

We tackle the issue with a novel design of discrete arrays (our term for arraylets). Our design delivers performance as good as systems that support

[3] The JIT compiler may alternatively generate literal pool access instructions.

contiguous arrays only. It differs from [7] in that discrete arrays are allocated only when the heap is under pressure, and discrete parts are not necessarily with equivalent size. Figure 4 illustrates our design of discrete array layout. It consists of a major part (the first part) and several tailing parts. We employ a greedy allocation strategy to allocate the major part as large as possible. Consequently most array accesses may occur in the major part. Then we allocate the rest of the requested space into many small free segments. The size of these segments is fixed so that the index of an array element, if falling into tailing parts, can be translated into the actual address within bounded time. This layout design, combined with a new array access code sequence, guarantees that the majority of accesses to a discrete array are nearly as fast as those to a contiguous array.

To support discrete arrays, the array layout is revised as follows:

```
struct array_type {
    obj_info header;
    uint32 length;        /* count of array elements */
    uint32 part1_len;     /* a new field */
};
```

Compared with conventional array layouts, this design adds a new field `part1_len`. The field has different usages for contiguous and discrete arrays. For contiguous arrays, `part1_len` equals to the value of `length`; when it comes to discrete arrays, `part1_len` implies the count of array elements located in the major part. We argue that the additional space introduced by the new field (one word per array) is trivial, given that average array size tends to be large empirically.

We allocate contiguous arrays the same way as conventional systems, except that the `part1_len` needs to be set as the value of `length`. XAMM always prefers contiguous arrays to discrete arrays. If allocation of a contiguous array fails, XAMM resorts to allocating discrete arrays. Allocation of discrete arrays requires two steps: (1) Best-effort allocation for the major part and (2) subsequent allocations for equally-sized tailing parts. The user level view of a discrete array is the same as contiguous arrays, i.e. its visible part starts from the location depicted by the arrow *discrete array ref* in Figure 4. *Pointers area*, the shadowed slots at the opposite side of *discrete array ref*, stores pointers to tailing parts. We use these pointers to calculate addresses for elements that reside in the tailing parts.

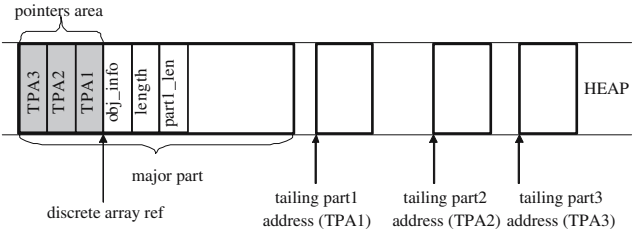


Fig. 4. A typical layout of discrete array

[7] adopts a uniform code sequence to access contiguous and discrete arrays: Query discrete parts information, map the index to its corresponding address, and finally load element from the specific address. We generate the code differently to make the common case path as fast as possible:

```
1  if (index >= array_ref->part1_len) {
2      if (index >= array_ref->length)
3          throw_out_of_bound_exception();
4      addr = discrete_array_map_index_to_addr(array_ref, index);
5  } else
6      addr = array_ref + sizeof(array_type) + index * sizeof(int);
7  R = [addr]
```

For contiguous arrays, `part1_len` equals to `length`. Accordingly the execution goes through the path 1-6-7, which results in comparable performance as conventional systems. For discrete arrays, array accesses are very probable to take place in the major part. In this case the execution path is still 1-6-7. But if `index` falls into the tailing parts, a slower path 2-4-7 is executed. The routine `discrete_array_map_index_to_addr` translates the index into an address, the complexity of which is determined by the structure of pointers area.

Reference arrays are not eligible for discrete array allocation, since we do not want to complicate the process of scanning reference arrays, which could be harmful for GC pause time. This design is acceptable because allocation of large reference arrays happens rarely.

Finally, we allow different survival strategies for a discrete array when its lifetime spans multiple GC cycles:

- A discrete array may survive a GC without changing its layout.
- A discrete array may regress to a contiguous array or another discrete array with its major part enlarged, based on the space availability after GC.
- A discrete array may regress to another discrete array with the major part enlarged or shrunken to a new size, if the size allows that more accesses hit into the major part.

3.4 Location Bits Management

Location bits, `locbits`, is a bit vector with the size proportional to the heap size. Each bit in the space is mapped to a unique heap address where a legal object may reside (we allocate location bits for all 4-aligned addresses). Conventionally `locbits` is used in both allocation phase and GC cycles: Once an object is successfully allocated, its corresponding location bit is set[4]. Computing the root set of live references, called *root set enumeration*, depends on `locbits` to screen out valid object addresses from spurious references on the stack. When GC ends, those bits associated with dead objects are unset. The complexity of clearing dead bits is O(heap_size). It is usually time consuming.

Location bits imposes extra space overhead[4]. We let `locbits` and mark bits share the same space. The live periods of the two spaces do not overlap. This fact

[4] That's why the `locbits` is referred as allocation-bits in [4].

implies that one space, `M/Abits`, is physically sufficient if we carefully maintain its semantics during phase transitions. We define the phased behavior as follows:

1. Assume `M/Abits` is initially cleared at the runtime bootstrapping phase.
2. During allocation phases, `M/Abits` acts as `locbits`.
3. GC cycle begins with the root set enumeration; the enumeration subsystem consults `M/Abits`, as `locbits` in fact, for pointer identification.
4. When the root set enumeration is over, the role of `M/Abits` is switched to mark bits (all unmarked initially). In the following mark-sweep phase, the bits associated with living objects are marked.
5. After the mark-sweep phase, `M/Abits` can be multiplexed for other purposes, for example, as the break table indexer to improve compaction performance.
6. As GC finishes, `M/Abits` is cleared again and those `locbits` for all retained objects are re-generated.

Though `M/Abits` solves the space issue, the management of `locbits` constitutes a considerable runtime overhead: We may continuously pay the cost of setting location bit during allocation, which generally results in more memory accesses and computation cycles. `M/Abits` has to be cleared over and over during phase transitions. What's more, the re-generation of `locbits` is hazardous in terms of GC pause time. XAMM addresses this issue using a lazy and selective approach — the essence of which is to set location bits on demand and only during the root set enumeration. The motivation behind is that `locbits` is only used in the root set enumeration, and usually we only need to query a very limited subset of it, thanks to relatively small number of spurious pointers. Therefore we can defer the setting of `locbits` till the enumeration phase, and generate only a small portion of the total location bits on demand.

The phased behavior of the new design does not change too much, except for Step 2, 3 and 6. We remove Step 2 so that the allocation time cost is completely dissolved. Step 6 is also eliminated, which effectively reduces the GC pause time. Step 3 becomes more complicated after planting in the lazy and selective location bits generation. The selective property of the algorithm relies, to a great extent, on our hierarchical heap organization, or the "divide and conquer" principle. Block is a self-sustained computation unit for most GC tasks. For a spurious pointer, its enclosing block is located first and we generate location bits only for objects resident in this block. With a proper division policy (block size vs. heap size), the overall complexity can be scaled down by order of magnitudes.

3.5 Selective Stack Clearing

Conservative GC must provide a mechanism to screen out spurious pointers. Prior arts, e.g. Boehm-Weiser Conservative GC[13], maintain segregated free lists, where pointer identification is relatively simple. As aforementioned, we don't adopt segregated lists based allocation. Instead we develop a new mechanism, namely location bits, to facilitate the screening process. In addition, our GC also uses alignment and type information to exclude specious pointers. For

instance, if a value happens to fall inside a char array but doesn't align with 2 (the size of a char element), we can deny it from being a legitimate pointer. To reduce the probability of mis-identification, we adopted the black-listing technique[11] to guarantee that certain stack slots, once treated as not roots, won't be mistakenly brought back into consideration by later allocations. Unfortunately this technique is not effective enough to avoid another type of false retaining: Sometimes the references to dead objects on the stack may continue to live through long cycles before they're overwritten.

We tackled this issue by injecting code that can proactively clear the stack at runtime. The code injection occurs on the fly. That is, XAMM decides where and when to do the code injection. If the heap is not under high pressure, no clearing code will be injected. But when XAMM runs out of space, the JIT compiler recompiles a certain portion of code to enable stack clearing code.

The ideal place for code injection should not sit in performance critical regions, but should make the clearing operation as effective as possible. We inject the code into the prologue or epilogue of methods or runtime stubs. To minimize its overhead, the injection heuristics are very carefully designed, to name a few:

- *Non-native methods*: Don't inject the code, that is, we simply assume they are performance sensitive.
- *Immediately-Returned native methods*: Inject the code into the epilogue of native method stubs, thus dead references can be removed immediately. The overhead is amortized by the long sequence of Java-to-native stub code.
- *Long-running native methods*: For example, `Object.wait()` or `Thread.sleep()`, we inject the code into the prologue and/or the epilogue of native method stubs, since these native methods don't return for a long period.
- *Runtime stubs*: Inject the code into both prologues and epilogues for some runtime stubs, e.g. GC invocation stub, since GC may grow the stack significantly and leave many object references on the stack.

The injected instructions dynamically check some conditions to determine whether to clear the stack. The condition we use for now is very simple: whether the current stack is shallow enough (the distance between the stack pointer and the bottom of the stack is smaller than some threshold). First, the check is very lightweight. The implementation for the check is just one instruction operating on registers, and the success rate for the check (i.e. clear the stack) is relatively small. So even though the clearing operation is costly, the average overhead from the code is quite small. Second, our experiments indicate the check is also effective in removing dead references. The rationale is that when the stack is shallow, it is likely that the program enters into a transitional state: a previous task just finishes, and a new one is about to start; If clearing the stack at this moment, the program thereafter is not affected by floating garbage. We now describe some scenarios that explain why this hypothesis is sound for cases.

- Consider a while loop in the `main` or `Thread.run()` method. For each iteration it deals with a heavy task that grows the stack intensively. At the end of each iteration, the stack goes back to its initial depth. At this moment, there

might be many dead references on the stack and it is a good time to clear them. A good example here is `treeadd` in the `Olden` benchmark suite.
- Imagine a system with several worker threads scheduled in an uneven manner. The worker threads wait on a queue after they finish some work on large objects. The references to these objects are left on the thread's stack for a long while until the thread is notified again. Our approach clears these references when the `Object.wait()` method traps into its native implementation. One of the EEMBC benchmarks, `parallel`, presents such a scenario.

Surprisingly the simple scheme can reduce the false retaining effectively without lowering performance. Our experimental results indicate that our approach can approximate the mostly accurate stack scanning approach in [10], but our design is much simpler in terms of VM and JIT compilers.

3.6 Prefetching

Prefetching is an efficient mechanism to hide memory latency. XAMM prefetches data intelligently according to the underlying system's prefetching parameters (e.g. cache line size, ideal prefetching distance). We classify memory behaviors into two categories:

- *Deterministic memory accesses*: In most cases it behaves as sequential striding. Typical activities in this category include sequential allocation, object scanning, and heap walking. Since the memory accesses are highly predictable, we can insert prefetching properly based on prefetching distance.
- *Non-deterministic memory accesses*: An exemplary activity is mark and scan. Mark stack, objects, mark bits, and object scanning information scatter throughout the heap. Albeit the typical mark-scan process exhibits irregular memory access patterns, its behavior can still be adapted to leverage prefetching technique —after `scan(object)` pushes items onto the mark stack, we know that the top of stack object is exactly the next candidate for marking and scanning, which could be the candidate for prefetching (as shown in the following code).

```
1  while (!mark_stack.empty()) {
2      object = mark_stack.pop();
3      /* mark(object); */
4      if (marked(object))
5          continue;
6      scan(object);
7      top_of_stack = mark_stack.peek();
8      prefetch(top_of_stack);
9      mark(object);
10     }
```

We motion `mark(object)` from Line 3 to 9. Generally the execution of `mark(object)` can cover the prefetching distance. Therefore when it comes to the next iteration, the data for that object are probably ready in the cache. However, the deferment of mark operation may require a large mark

stack to some extent, because objects in a circular data structure may be pushed onto the stack repeatedly. But it never turns out to be an issue in our experiments.

For the areas where prefetching cannot help, we resort to conventional cache-friendly designs: Allocate performance sensitive VM data structures aligned to cache line size, and aggregate temporally correlated fields to the same cache line.

4 Experimental Results

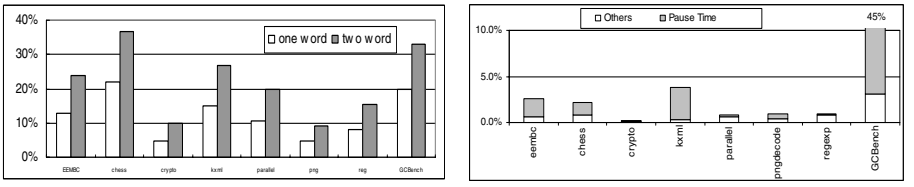
We run experiments on Intel XScale^{TM} (PXA255) development boards (CPU / Bus / Memory frequency: 333 / 166 / 84MHz; DCache: 32KB; ICache: 32KB; main memory: 64M; OS: ARM-Linux). XAMM is integrated into XORP runtime system. The configuration under test specifies a unified heap with 2MB[5]. As shown in Table 1, we selected six typical J2ME programs from EEMBC GrinderBench^{TM} benchmark suite and one well-accepted GC benchmark from Boehm. We will also present the data for a so-called "EEMBC" program, which runs the six individual EEMBC tasks serially as one application. Among these programs, Chess, Kxml and GCBench are sensitive to allocation performance; Kxml and GCBench are GC intensive.

Table 1. Benchmark Descriptions

Name	Description	Comments
Chess	Chess game	allocation intensive
Crypto	Cryptographic package	
Kxml	XML parser	allocation and GC intensive
Parallel	Multithreaded computation	large array allocations
PNG	PNG decoder	
RegExp	Regular expression package	
GCBench	Build trees	from Hans Boehm, allocation and GC intensive

First, we show the space and performance impacts from the compact header design. Figure 5(a) shows the percentage of the object header size to the total allocated size. The two-word header attributes a significant portion of space consumption (9%∼36%). Our compact header design reduces the overhead to 5%∼23%, which is a 4%∼13% saving of total allocated space. The space saving is not so distinct for Crypto and PNG because they allocate a lot of large objects. To translate the space saving into performance gain, we come up a two-word design with exactly the same layout as the compact one. The compact design can reduce GC occurrences significantly (13% for EEMBC and 55% for GCBench). Figure 5(b) shows the performance benefit from reduced memory usage. Two GC sensitive benchmarks, Kxml and GCBench, are improved by 4% and 45%

[5] We adapted some parameters for GCBench to fit it into a smaller heap.



(a) **(b)**

Fig. 5. Space and Performance: one-word vs. two-word

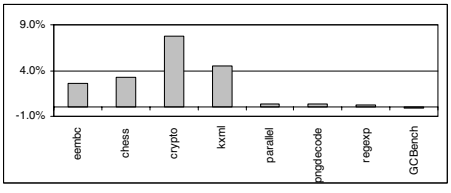


Fig. 6. Perf: 2-bit lock vs. thin lock

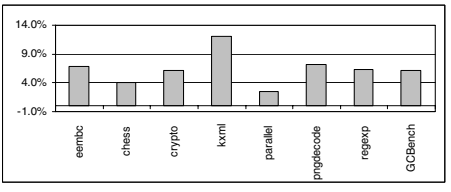


Fig. 7. Benefits from heap placement

respectively, largely thanks to the reduced GC pause time. Since GC sweeps the cache, the runtime behavior between GC cycles is also improved.

As an essential feature of the compact header, we claim that 2-bit lock has comparable performance as the typical thin-lock implementations[9]. To rule out the performance impact of the space discrepancy, we implement both lock mechanisms on a two-word header design. As depicted in Figure 6, 2-bit lock introduces on average less than 2.5% of performance degradation for EEMBC programs. 2-bit lock does not behave well if the program experiences many recursive locks. For recursive locks, 2-bit lock incurs the overhead of lock inflation, while thin lock does not. Thus Crypto has a 7% slowdown. Though Parallel indicates that 2-bit lock performs well compared to thin-lock, we need more studies by evaluating it in versatile multi-threaded environments.

Figure 7 shows the performance improvement when lowering the heap. For most programs we have performance benefits ranging from 2.5% to 7%. Surprisingly Kxml can be improved by 12%. The reason is that Kxml opens up many opportunities for the JIT compiler to generate good code. For instance, Kxml has large numbers of invocations from JIT'ed to VM helper routines, such as allocation, locking and type checking. Our experiments found that the optimized version reduces dynamic indirect calls by 300%. Kxml also frequently loads constant strings whose addresses can be expressed with very short instruction sequences if heap placement is enabled.

Lazy and selective location bits management improves the allocation performance and reduce the GC pause time. As shown in Figure 8, the raw performance improvement varies from 2% to 14% for all programs. We further break down the improvement into two categories: faster allocation and smaller pause time. For allocation intensive benchmarks such as Chess, Kxml and GCBench, 4%∼7% performance gain is obtained when we do not need to set location bits

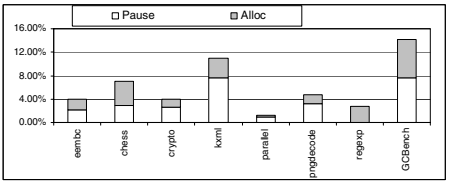


Fig. 8. Perf improvement from lazy and selective location bits management

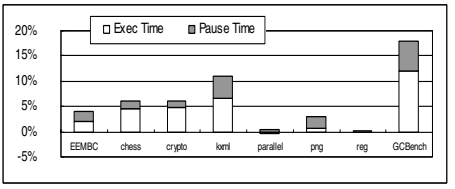


Fig. 9. Perf. & pause time improved by prefetching

per allocation request. The GC pause time is also significantly reduced because we don't have to walk the heap to recover the location bits for live objects. This mechanism offers a good payback for Kxml and GCBench: we gain about 7% better performance from the reduced GC pause time. This approach is very beneficial when the average pause time, rather than raw performance, is highly preferred. Our experiments show that it can reduce the average pause time by 4%∼25%.

Prefetching effectively reduces load penalty when memory accesses are predictable. Figure 9 depicts the performance improvement from prefetching. Kxml and GCBench are the two benchmarks that benefit most from prefetching: the total execution time is reduced by 11% and 18% respectively. Again, we break down the reduced execution time into two categories: GC pause time and non-GC time (mainly from cache-friendly allocation). Though non-GC time may attribute to a major portion of the total reduced execution time for most benchmarks, GC pause time is usually more crucial for real-world applications. We observed that our prefetching mechanism can curtail GC pause time by 3.6%∼22%, which could lead to significantly better user experience. We studied the performance gain with PMU (Performance Monitoring Unit) data, and found that it mainly comes from reduced cache misses.

It's highly debatable whether to deploy a conservative GC on memory constraint devices. At the bright side, conservative GC simplifies VM and JIT design; but the dark side is that it leads to floating garbage. In extreme cases, the conservatism may impair the system's availability when exposed to Denial-Of-Service (DoS) attacks. We developed some innovative mechanisms (e.g. selective stack clearing) to minimize the negative impacts from conservative GC. To quantify the effectiveness of these mechanisms, we implemented a mostly accurate GC (we call it MAG, which is similar to [10]) and compared its results with our ECG. The mostly accurate GC minimizes the conservatism by scanning the stack with a mostly accurate fashion, combining accurate JIT enumeration and conservative native stack scanning. The cost is that the JIT compiler must be cooperative in bookkeeping exact GC information at GC points. But the gain is that the simplicity in VM design is still there. Table 2 presents the comparison between MAG and ECG. The two methods cause the same number of GC (the second column). We measured the average amount of retained objects and the average size of retained memory, and for both sets of data, we calculated the percentage of less retained data with ECG (negative numbers indicate that ECG

Table 2. Mostly Accurate GC (MAG) vs. Enhanced Conservative GC (ECG)

		Avg. Retained Obj. Count			Avg. Retained Memory		
Program	#GC	MVG	ECG	ECG vs. MVG	MVG	ECG	ECG vs. MVG
EEMBC	215	2037	2177	6.45%	124889	126411	1.20%
Chess	34	756	729	-3.67%	30286	29310	-3.33%
Crypto	24	406	408	0.53%	52742	51517	-2.38%
Kxml	55	3485	3456	-0.82%	137933	136837	-0.80%
Parallel	9	466	460	-1.28%	129762	128546	-0.95%
PNG	30	743	742	-0.07%	87037	87021	-0.02%
RegExp	2	644	647	0.46%	19556	19600	0.22%
GCBench	12	8858	8855	1.71%	427762	427778	0.00%

Table 3. Overall Performance Improvement

Program	Baseline	Optimized	Improved
EEMBC	87791	80738	8.4%
Chess	19331	17759	8.9%
Crypto	14432	13389	7.8%
Kxml	14586	11650	25.2%
Parallel	13253	13104	1.1%
PNG	11065	10263	7.8%
RegExp	16422	15391	6.7%
GCBench	1889ms	1212ms	55.9%

is better). From the table, we concluded that ECG achieves comparable accuracy as MAG. ECG doesn't perform well for EEMBC in terms of the number of retained objects. This is reasonable since EEMBC is a collection of applications with different behaviors, which defeats our relatively simple heuristics. According to [10], the falsely retained space can be reduced by 99.7% maximally; in other words, MAG is nearly as accurate as a real accurate GC. We believe that XAMM can perform equally well with ECG on small memory systems.

The sum of all the above techniques has a comprehensive impact on the system-wide behavior. Table 3 presents the performance improvement over a baseline version which deliberately turns off all the optimizations. For most EEMBC programs except Parallel, our optimizations can reduce the execution time by 7% to 25%. And because GCBench spends most of its time in allocation and GC, the improvement amounts to 55.9%.

5 Related Work

Memory management on memory/computation constraint devices has been an interesting research area in recent years. Memory footprint, pause time and performance are the hottest topics in this area. MC^2 (Memory-Constrained

Copying)[20], developed by Narendran et al., is a representative research considering these design spaces all together. It is a copying generational GC that can meet soft real-time requirements on memory-constrained systems. Bacon et al.[6] also discussed how to deliver real-time response with controlled space consumption. Energy research from the view point of GC[16] has also emerged recently.

How conservative GC deals with these aspects is another interesting research area. Endo et al.[17] described several approaches to reduce pause time of conservative GC while keeping the execution overhead low. Boehm[12] discussed some ways of prefetching to reduce cache misses in his classical conservative GC. Barabash et al.[10] tried a different approach, mostly accurate stack scanning, to minimize conservatism by combining cooperative JIT enumeration and uncooperative native stack scanning. We implemented a similar approach to compare its effectiveness with our selective stack clearing.

Designing a compact object layout is one approach to reduce memory expenditure. Bacon et al.[8] pioneered one-word object model design by handling hashcode and lock state specially. State-of-the-art one-word header designs, such as indexed object model[8] and Monty[3], have to compromise on some good properties that full-fledged header provides, and may increase maintenance complexity.

Among those challenges originated from small memory budget, large object allocation is always difficult. Section 3.3 compared our discrete array design with object compression[15] and arraylets[6].

Ali et al.[5] proposed *Mississippi Delta* technique, which is another example that assists JIT code generation (prefetch injection) with GC's knowledge.

6 Conclusions

This paper presents a set of techniques to build an automatic memory management system, with balanced considerations of performance, memory consumption and response time. Among the techniques, most are general enough to use in systems other than XORP. We can also easily apply the platform-specific optimizations to more general environments. Also, some techniques open up the possibility of using memory management to instruct other components' optimizations, which effectively expands XAMM's role. The selective stack clearing can effectively remove falsely retained garbage for simple programs, but not for ones with versatile behaviors. In the future, we will try a more proactive and precise approach. That is, JIT compilers will generate fine-grained clearing instructions immediately after references become dead, based on a learning model from XAMM. Our experimental results also demonstrate the feasibility to achieve high performance without losing memory efficiency. We will continue investigating innovative technologies on XAMM especially in the following aspects: controlled response time, energy consumption, new GC algorithms and implementations, and XAMM's interactions with other components.

References

1. Jikes Research Virtual Machine (RVM). At URL: www-124.ibm.com/developerworks/oss/jikesrvm/.
2. Open Runtime Platform (ORP) from Intel Corporation. At URL: http://orp.sourceforge.net.
3. Project Code-named "Monty": A High Performance Virtual Machine for the Java^{TM} Platform for Small Devices. At URL: http://servlet.java.sun.com/javaone/sf2002/conf/sessions/display-2133.en.jsp.
4. Sensible Sanitation – Understanding the IBM Java Garbage Collector. At URL: http://www-106.ibm.com/developerworks/ibm/library/i-garbage1/.
5. Adl-Tabatabai, A.-R., Hudson, R. L., Serrano, M. J., and Subramoney, S. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of PLDI'04*, pp. 267–276.
6. Bacon, D. F., Cheng, P., and Rajan, V. T. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for java. In *Proceedings of LCTES'03*, pp. 81–92.
7. Bacon, D. F., Cheng, P., and Rajan, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of POPL'03*, pp. 285–298.
8. Bacon, D. F., Fink, S. J., and Grove, D. Space- and time-efficient implementation of the java object model. In *Proceeding of ECOOP'02*, pp. 35–46.
9. Bacon, D. F., Konuru, R., Murthy, C., and Serrano, M. Thin locks: Featherweight synchronization for Java. In *Proceedings of PLDI'98*, pp. 258–268.
10. Barabash, K., Buchbinder, N., Domani, T., Kolodner, E. K., Ossia, Y., Pinter, S. S., Shepherd, J., Sivan, R., and Umansky, V. Mostly accurate stack scanning. In *Proceedings of the Java^{TM} Virtual Machine Research and Technology Symposium 2001 (JVM'01).*
11. Boehm, H.-J. Space efficient conservative collection. In *Proceedings of PLDI'93*, pp. 197–206.
12. Boehm, H.-J. Reducing garbage collector cache misses. In *Proceedings of ISMM'01* (2001), pp. 59–64.
13. Boehm, H.-J., and Weiser, M. Garbage collection in an uncooperative environment. *Software Practice & Experience 18*, 9 (1988), 807–820.
14. Bracha, G., Gosling, J., Joy, B., and Steele, G. *The Java Language Specifications, Second Edition.* Addison Wesley, Jun.2000.
15. Chen, G., Kandemir, M., Vijaykrishnan, N., Irwin, M. J., Mathiske, B., and Wolczko, M. Heap compression for memory - constrained Java environments. In *Proceedings of OOPSLA'03*, pp. 282–301.
16. Chen, G., Shetty, R., Kandemir, M. T., Vijaykrishnan, N., Irwin, M. J., and Wolczko, M. Tuning garbage collection for reducing memory system energy in an embedded java environment. In *ACM Transactions on Embedded Computing Systems (TECS)* (2002), vol. 1, pp. 27–55.
17. Endo, T., and Taura, K. Reducing pause time of conservative collectors. In *Proceedings of ISMM'02*, pp. 119–131.
18. Haddon, B. K., and Waite, W. M. A compaction procedure for variable length storage elements. In *Computer Journal* (1967), vol. 10, pp. 162–165.
19. Jones, R., and Lins, R. *Garbage collection: algorithms for automatic dynamic memory management.* John Wiley & Sons, Inc., 1996.
20. Sachindran, N., Moss, J. E. B., and Berger, E. D. Mc^2: High-performance garbage collection for memory-constrained environments. In *Proceedings of OOPSLA'04.*

Memory-Centric Security Architecture

Weidong Shi, Chenghuai Lu, and Hsien-Hsin S. Lee

College of Computing, School of Electrical and Computer Engineering,
Georgia Institute of Technology, Atlanta, GA 30332

Abstract. This paper presents a new security architecture for protecting software confidentiality and integrity. Different from the previous process-centric systems designed for the same purpose, the new architecture ties cryptographic properties and security attributes to memory instead of each individual user process. The advantages of such a memory centric design are many folds. First, it provides a better security model and access control on software privacy that supports both selective and mixed tamper resistant protection on software components from heterogeneous sources. Second, the new model supports and facilities tamper resistant secure information sharing in an open software system where both data and code components could be shared by different user processes. Third, the proposed security model and secure processor design allow software components protected with different security policies to inter-operate within the same memory space efficiently. Our new architectural support requires small silicon resources and its performance impact is minimal based on our experimental results using commercial MS Windows workloads and cycle based out-of-order processor simulation.

1 Introduction

Recently, there is a growing interest in creating tamper-resistant/copy protection systems that combine the strengths of security hardware and secure operating systems to fight against both software attacks and physical tampering of software [2,3,6,7,11,12,13]. Such systems aim at solving various issues in the security domain such as digital rights protection, virus/worm detection, intrusion prevention, digital privacy, etc. For maximum protection, a tamper-resistant/copy protection system should provide protection against both software and hardware based tampering including duplication (copy protection), alteration (integrity and authentication), and reverse engineering (confidentiality).

Many the aforementioned copy protection systems achieve protection by encrypting the instructions and data of a user process with a single master key. Although such closed systems do provide security for software execution, they are less attractive for real world commercial implementations because of the gap between a closed tamper-resistant/copy protection system and real world applications that are mostly multi-domained where a user process often consists of components coming from heterogeneous program sources with distinctive security requirements. For instance, almost all the commercial applications use statically linked libraries and/or dynamically linked libraries (DLL). It is quite

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 153–168, 2005.

natural that these library vendors would prefer a separate copy protection of their own intellectual properties decoupled from the user applications. Furthermore, it is also common for different autonomous software domains to share and exchange confidential information at both the inter- and intra- process levels. The nature of de-centralized development of software components by different vendors makes it difficult to enforce a process centric protection scheme.

Traditional capability-based protection systems such as Hydra [1] and CAP [4] although provide access control on information, they were not designed for tamper resistance to prevent software duplication, alternation, and reverse engineering. Specifically, they do not address how access control interacts with other tamper resistant protection mechanisms such as hardware based memory encryption and integrity verification.

In this paper, we present a framework called *MEmory-centric Security Architecture* or *MESA* to provide protection on software integrity and confidentiality using a new set of architectural and OS features. It enables secure information sharing and exchange in a heterogeneous multi-domain software system. The major contributions of our work are summarized as follows:

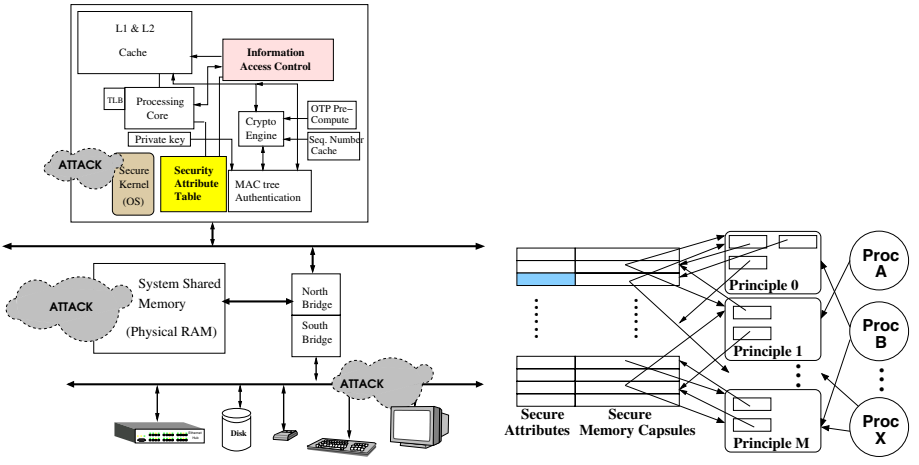
- We presented and evaluated a unique memory-centric security model for tamper resistant secure software execution. It distinguishes from the existing systems by providing better support for inter-operation and information sharing among software components in an open heterogeneous multi-domain software system.
- We introduced architecture innovations that allow efficient implementation of the proposed security model. The proposed secure processor design incorporates and integrates fine-grained access control of software components, rigorous anti-reverse engineering, and tamper resistance.
- We discussed novel system mechanisms to allow heterogeneous program components to have their own tamper resistant protection requirements and still to be able to inter-operate and exchange information securely.

The rest of the paper is organized as follows. Section 2 introduces MESA which is extended in Section 3 that details each MESA component. Evaluation and results are in Section 4. Discussion of related work is presented in Section 5 and finally Section 6 concludes.

2 Memory-Centric Security Architecture

In this section, we overview our *Memory-centric Security Architecture* (MESA). Using novel architectural features, MESA enables high performance secure information sharing and exchange for a multi-security domain software system. Figure 1(a) shows MESA and its operating environment.

Now we present MESA from system perspective. One critical concept of MESA is the *memory capsule*. A memory capsule is a virtual memory segment with a set of security attributes associated with it. It is an information container that may hold either data or code or both. It can be shared by multiple processes.



(a) Memory-centric Security Model

(b) Principles and Security Memory Capsules

Fig. 1. Memory-centric Security Architecture (MESA)

For example, a DLL is simply a code memory capsule. A set of security attributes are defined for each memory capsule besides its location and size. These security attributes include security protection level, one or more symmetric memory encryption keys, memory authentication signature, accesses control information, etc. For each process, the secure OS kernel maintains a list of memory capsules and their attributes as process context. During software distribution, software vendors encrypt the security attributes associated with a memory capsule using the target secure processor's public key. The secure processor authenticates and extracts the security attributes using the corresponding private key. A secure processor never stores security attributes in the exposed physical RAM without encryption protection.

Another important concept is *principle*. A principle is an execution context smaller than a process. It is defined as the execution of secure code memory capsules having the same security property within a user process. Principles are associated with memory capsules. They can be considered owners of memory capsules. Based on the associated principles and security protection levels, access control of memory capsules can be carried out. An active principle is a currently executing principle. When an active principle accesses some memory capsule, the access will be checked. If the active principle is allowed to access the memory capsule, the access will be granted, otherwise, security exception of access violation will be raised. Note that the *principle* in MESA is different from the *principle* defined in capability system such as [1] and [4].

MESA allows different keys been used to protect separate memory capsules and enforces access control during program execution. However, the scenarios of fine-grained dynamic information sharing frequently happens during program execution. For example, 1) An application calls OS services such as fwrite, fread

and passes pointers to data buffer owned by the application; 2) An application calls OS and system services to get a pointer to data structures owned by the OS or system libraries; 3) A principle calls a routine of another principle and the caller requires the callee to either operate or modify some data content in a data memory capsule it owns.

2.1 Secure Capsule Management

Most functionality of the secure capsule management is achieved by a secure OS kernel. Among the major services provided by the secure kernel are, process and principle creation, principle authentication, principle access control.

First, during a process creation, the secure OS kernel will create a list of memory capsules associated with the process. The secure kernel creates application process from the binary images provided by software vendors. Each binary image may contain one or more protected code and data sections, each one with its own security attributes. Security attributes of binary images for the application, middleware, and system shared libraries are set independently by their corresponding vendors. The secure kernel creates a secure memory capsule context based on the binary images. Each memory capsule represents an instantiation of either a code module or data module and is uniquely identified with a randomly generated ID. For DLLs, a different capsule is created with a different ID when it is linked to a different process. However note that the code itself is not duplicated. It is simply mapped to the new process's memory space with a different capsule entry in the capsule context table.

Execution of a process can be represented as a sequence of executing principles. Heap and stack are two types of dynamic memory that a principle may access. Privacy of information stored in the heap and stack is optionally protected by allocating private heap and stack memory capsule to each principle. Another choice is to have one memory capsule to include both protected code image and memory space allocated as private heap and stack. When execution switches to a different principle, the processor's stack pointer is re-loaded so that it will point to the next principle's private stack. Details of private stack are presented in the subsection of intra-process sharing.

With the concept of private heap, it comes the issue of memory management of private heaps associated with each principle. Does each principle require its own heap allocator? The answer is no. Heap management can be implemented in a protected shared system library. The key idea is that with hardware supported protection on memory integrity and confidentiality, the heap manager can manage usage of each principle's private heap but cannot tamper its content.

To provide a secure sharing environment, support for authenticating principles is necessary. MESA supports three ways of principle authentication. The first approach is to authenticate a code memory capsule and its principle through a chain of certification. A code memory capsule could be signed and certificated by a trusted source. If the certification could be verified by the secure kernel, the created principle would become a trusted principle because it is certified by a trusted source. Another approach is to authenticate a principle using a public

key supplied by software vendors. This provides a way of private authentication. The third way is to certify principle using secure processor's public key. For example, application vendors can specify that the linked shared libraries must be certified by the system vendors such as Microsoft and the middle-ware image must be certified by the known middle-ware vendors. Failure of authenticating a code image will abort the corresponding process creation.

2.2 Intra Process Sharing

There are many scenarios that pointers need to be shared by multiple principles. One principle calls a routine of another principle and passes pointers to data belonging to a confidential data memory capsule. In this scenario, information security will be violated if the callee's function attempts to exploit the caller's memory capsule more than what is allowed. For instance, the caller may pass a memory pointer, *mp* and length *len*, to the callee and restricts the callee to access only the memory block *mp[0, len-1]*. However, the callee can spoil this privilege and try to access to the memory at *mp[-1]*. This must be prevented. Passing by value may solve the problem but it is less desired because of its cost on performance and compatibility with many popular programming models. MESA facilitates information sharing using explicit declaration of shared subspace of memory capsules. The owner principle of a memory capsule is allowed to add more principles to share data referenced by a pointer that points to its memory capsule. However, the modification is made on a single pointer basis. The principle can call sec_add_sharing_ptr(addr, size, principle_p) intrinsic to declare

Intrinsic	Parameters	Explanation
sec_malloc(s,id)	s: *size* ; id: *principle id*	allocate memory from a principle's private heap
sec_free(p)	p: *memory pointer*	free memory to its owner's private heap
sec_swap_stack (addr)	addr: *address*	switch the active stack pointer to another principle's private stack. Addr points to a location of the target principle. Save <active stack pointer, active principle id> to the stack context table
sec_get_id (name)	name: *capsule name*	get id of a principle (secure kernel service)
sec_push_stack_ptr()		read the current executing principle's stack pointer from the stack context table and push it into its private stack
sec_save_ret_addr (addr)	addr: *address*	assign addr to a return address register (RAR)
sec_return()		assign RAR to PC and execute
sec_add_sharing_ptr (p, s, id, rw)	p: *pointer;* s: *size* ; id: *principle id* ; rw: *access right*	allow target principle (id) to access memory region [p, p+s) with access right rw, return a security pointer that can be passed as function parameter (secure kernel service)
sec_remove_ptr (p)	p: *security pointer*	remove access right granted to security pointer p
sec_save_security_ptr (reg, addr)	reg: *register holding security pointer* ; addr: *address*	save security pointer to memory
sec_load_security_ptr (reg, addr)	reg: *register holding security pointer* ; addr: *address*	load security pointer from memory

Fig. 2. MESA Security Intrinsics

```
// application: my_app
// middleware: my_middle as DLL
void* p;
unsigned int my_id, middle_id;
security void* sp;
int ret;

my_id = sec_get_id("my_app");
p = sec_malloc(0x20, my_id);
...
middle_id = sec_get_id("my_middle");
sp = sec_add_sharing_ptr(p, 0x20,
      middle_id, RD|WR);
ret = my_middle_foo(sp, 0x10);
sec_remove_ptr(sp);
...
sec_free(p);
```

(a) Printer Sharing

```
// CALLER SIDE
push ebp //save stack frame pointer
sec_swap_stack addr_of_my_middle_foo
// stack pointer switched to the callee's stack
// caller's esp saved to the context table
push 0x10; // input parameter
call my_middle_foo
//push return PC to the callee's stack
pop ebp //get stack frame pointer back

// CALLEE SIDE
sec_push_stack_ptr
ebp = esp //ebp stack frame pointer
...
r1 = [ebp + 4] //return address
sec_save_ret_addr r1 //save caller's return
esp = [ebp]
sec_swap_stack r1
// restore stack pointer to the caller's stack
// callee's esp saved to the context table
sec_return //return to the caller
//return to the caller by loading return PC
```

(b) Securely Maintain Correct Stack Behavior

Fig. 3. Cross Principle Function Call

that *principle_p* is allowed to access the data in the range [addr, addr+size]. The declaration is recorded in a pointer table. Dynamic access to the memory capsule is checked against both the memory capsule context and the pointer table. After the pointer is consumed, it is removed from the pointer table by another intrinsic sec_remove_ptr(addr). This mechanism allows passing pointers of either private heap or private stack during a cross-principle call.

Aside from the above two intrinsics, MESA also proposes other necessary intrinsics for managing and sharing secure memory capsules as listed in Figure 2. Note that these security intrinsics are programming primitives not new instructions. Although a hardware implementation of MESA can implement them as CISC instructions, yet it is not required.

Figure 3(a) illustrates how to securely share data during a function call using MESA's security intrinsics. There are two principles. One is "my_app" and the other is a middleware library called "my_middle". "my_app" allocates a 32-byte memory block from its own private heap by calling sec_malloc(0x20, my_id). Then it declares a sharing pointer p using intrinsic sec_add_sharing_ptr that grants principle "my_middle" read/write access to the memory block pointed by p. The intrinsic call returns a security pointer, sp. After declaring the security pointer, principle "my_app" calls my_middle_foo() of principle "my_middle", passes the security pointer and another input argument. Inside my_middle_foo(), codes of principle "my_middle" can access the private memory block defined by the passed security pointer sp. After the function returns control back to principle "my_app", "my_app" uses intrinsic sec_remove_ptr to remove the security pointer thus revoking "my_middle"'s access to its private memory block. In addition to passing shared memory pointers, during the cross-principle call, program stack has to

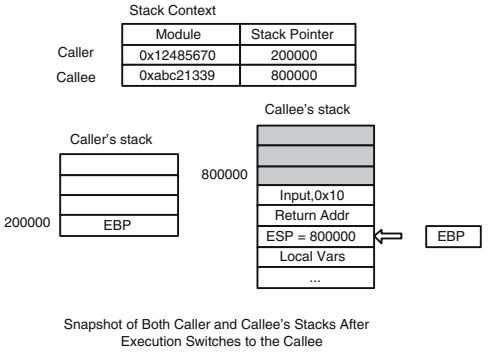


Fig. 4. Snapshot of Stacks After a Cross-Principle Call

securely switch from the caller's private stack to the callee's private stack using stack-related security intrinsics.

To show how private stacks are protected during cross-principle call, we show what happens at assembly level when "my_app" calls my_middle_foo() and the exit code of my_middle_foo() in Figure 3(b). The assembly code uses x86 instructions and MESA intrinsics. In the example, the caller switches stack pointer to the callee's private stack using sec_swap_stack intrinsic. Input addr is either a function entry address or return address if the intrinsic is used to switch stack pointer from the callee's private stack to the caller's. MESA maintains a table of stack pointer context for all the running principles. When sec_swap_stack(addr) is executed, it will save the current active stack pointer as a <principle, stack pointer> pair and set the active stack pointer to the target principle's. Then it pushes values to the callee's stack memory capsule. When the callee's stack capsule requires information to be encrypted, the pushed stack values will be encrypted using the callee's key. Since the caller and the callee use different stacks, to maintain compatibility with the way how stack is used for local and input data, the callee uses sec_push_stack_ptr intrinsic at the function entry point to push its stack pointer from the context table into its private stack. Figure 4 shows both the caller and the callee's stacks after the discussed cross-principle call. When execution switches back from "my_middle" to "my_app", the callee copies the returned value to the caller's stack capsule. Since the return PC address is saved in the callee's private stack (happens during execution of function call instruction after stack swap), the callee has to read the return address and put it into a temporary register using sec_save_ret_addr. Then the callee switches the stack pointer to the caller's. Finally, the callee executes sec_return intrinsic that assigns the returned address stored in the temporary register to the current program counter. In the case where a large amount of data need to be returned, the caller could reserve the space for the returned value on its stack by declaring a security pointer pointing to its stack and passes it to the callee.

Note that MESA protects against tampering on the target principle's stack by only allowing one principle to push values to other principle's stack. A principle can not modify another principle's stack pointer context because only the owner

principle can save the active stack pointer as its stack pointer context according to the definition of sec_swap_stack. Explicitly assigning values to the active stack pointer owned by a different principle is prohibited.

2.3 Inter Process Sharing - Shared Memory

MESA supports tamper-resistant shared memory through access control by the secure kernel. In MESA, each memory capsule can be owned by one or more principles as shown in Figure 1(b). Access rights (read/write) to a secure memory capsule could be granted to other principles. For secure sharing, the owner principle specifies the access right to be granted only to certain principles that are authenticated. Using this basic secure kernel service, secure inter-process communication such as secure shared memory and secure pipe can be implemented. For example, assume that P1 and P2 are two user principles belonging to different processes that want to share memory in a secure manner. Principle P1 can create a memory capsule and set a sharing criteria by providing a public authentication key. When principle P2 tries to share the memory created by P1, it will provide its credential, signed certificate by the corresponding private key to the secure kernel. The secure kernel then verifies that P2 can be trusted and maps the capsule to the memory space of P2's owning process.

3 Architectural Support for MESA

In this section, we discuss the architectural support for MESA. Inside a typical secure processor, we assumed a few security features at the micro-architectural level that incorporate encryption schemes as well as integrity protection based on prior art in [2,7,6]. In addition, we introduce new micro-architecture components for the MESA including a *Security Attribute Table (SAT)*, an *Access Control Mechanism (ACM)*, and a *Security Pointer Buffer (SPB)*. Other new system features are also proposed to cope with MESA in order to manage the security architecture asset.

3.1 Security Attribute Table

Secure memory capsule management is the heart of MESA. Most of the functionality for secure memory capsule management is implemented in the secure OS kernel that keeps track of a list of secure memory capsules used by a user process. Security attributes of frequently accessed secure memory capsules are cached on-chip in a structure called Security Attribute Table (SAT). Figure 5 shows the structure of a SAT attached to a TLB. Each entry stores a set of security attributes associated with a secure memory capsule. The crypto-key of each SAT entry is used to decrypt or encrypt information stored in the memory capsule. The secure kernel uses intrinsic sec_SATld(addr) to load security attributes from the memory capsule context stored in memory to SAT. The crypto-keys in the memory capsule context are encrypted using the secure processor's public key. The secure processor will decrypt the keys when loading them into the SAT.

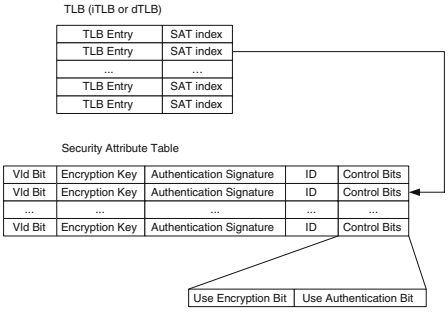


Fig. 5. Security Attribute Table (SAT) and TLB

A secure memory capsule could be bound to one or many memory pages of a user process. When the entire user virtual memory space is bound to only one secure memory capsule, the model is equivalent to a process-centric security model. As Figure 5 shows, each TLB entry has an index to SAT for retrieving the security attributes of its corresponding memory page. During context switches, the secure kernel authenticates the process's memory capsule context first, then loads security attributes into SAT. SAT is accessed for each external memory access. For a memory access missing in the cache, data in the external memory will be brought into the cache, decrypted using the encryption key in the SAT and its integrity verified against the root memory authentication signature, also stored in the SAT using hash tree [7]. On-chip caches maintain only plaintext. SAT is also accessed when data is evicted from the on-chip caches. The evicted data will be encrypted using keys stored in the SAT and a new memory capsule root signature will be computed to replace the old root signature.

If the required security attributes could not be found in the SAT, a SAT fault is triggered and the secure kernel will take over. First, the secure kernel flushes the cache, then load the required security attributes into the SAT. The SAT indexes stored in the TLB are also updated accordingly.

3.2 Access Control Mechanism

Efficient hardware-based access control plays a key role for protecting memory capsules from being accessed by un-trusted software components. It is important to point out that having software components or memory capsules encrypted does not imply that they can be trusted. Adversaries can encrypt a malicious library and have the OS to load it into an application's virtual space. The encrypted malicious library despite encrypted can illegally access confidential data.

Access management is achieved by associating principles with memory capsules they can access. The information is stored in a table. Based on the security requirements, the secure processor checks the table for every load/store operation and the operation is allowed to be completed only if it does not violate any access restraint. To speed up memory operations, frequently accessed entries of the rule table can be cached inside the processor by an *Access Lookup Cache*

(ALC). Entries of ALC are matched based on the current executing principle's ID and the target memory capsule's ID. To avoid using a CAM for ALC implementation, executing principle's ID and the target memory capsule's ID can be XORed as the ALC index shown in Figure 6. For a memory operation and ID tuple <running principle ID, memory capsule ID, rd, wr>, if a matching ALC entry can not be found, an ALC miss exception will be triggered and program execution will trap into the secure kernel. The secure kernel will check the complete access control table to see whether a matching entry can be found there. If an entry is found, it will be loaded into the ALC. Failure to find an entry in both the ALC and the complete access control table implies the detection of an illegal memory operation. Upon detection of such an operation, the secure kernel will switch to a special handling routine where proper action will be taken either terminating the process or nullifying the memory operation.

To minimize performance impact of ALC lookup on cache access, ALC checking can be conducted in parallel with storing information into the cache, thus incurs almost no performance loss since stores are typically not on the critical path. As shown in Figure 6, there is one pass bit in each cache line indicating whether the stored data yet passes access control security checking. When data is written to a cache line, the bit will be clear. When ALC lookup returns a hit, the bit will be set. To guarantees that a secure memory capsule is not updated by instructions whose principles do not have write access right, cache lines with the checked bit clear are inhibited from being written back to the memory. Furthermore, read access to a cache line with the check bit set is stalled until the bit is set. The performance impact of the above mechanism on cache access is minimal because as our profiling study indicates the interval between the same dirty line access is often long enough to hide the latency of ALC lookup.

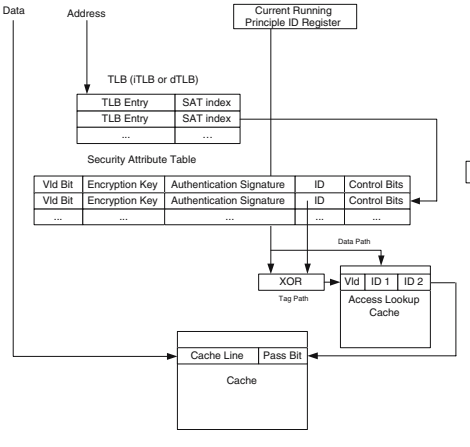


Fig. 6. Information Access Control Mechanism

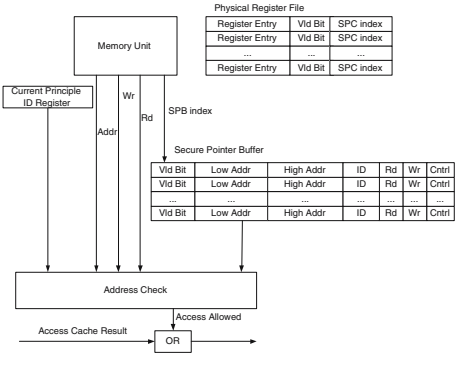


Fig. 7. Security Pointer Buffer

3.3 Security Pointer Buffer

MESA requires tracking shared pointers in a security pointer table. To improve performance, values of the pointer table are cached in an on-chip *Security Pointer Buffer (SPB)* shown in Figure 7. Each entry records the low and high address of the shared memory region and the principle ID that is granted temporary access right. A new programming data type, security pointer is required. The difference between a security pointer and a regular pointer is that aside from the address value, security pointer also maintains other information for identifying the declared pointer such as an index to the SPB, see Figure 7. If a memory address held by a register is added to the SPB, the involved register will be tagged with the index of the added entry in the SPB and a security pointer valid bit is set. When the register value is assigned to another register, the index is also passed. When a null value is assigned to the register, the valid bit is cleared. When address stored in a security pointer is used to access memory and its valid bit is set, the associated index is used to retrieve the corresponding entry in the SPB. The memory address is compared against the memory range stored in the SPB, and ID of the running principle is also compared with the ID stored in the corresponding SPB entry. If the memory address is within the range, the principle IDs are identical, and the type of access (read/write) also matches, the memory reference based on the security pointer is accepted. Otherwise, a security pointer exception will be raised.

Security pointer exception is raised only when memory access is issued using a valid security pointer and there is a mismatch in the SPB. The exception is handled by the secure kernel. There are two possible reasons for a SPB access failure. First, the entry may be evicted from the SPB to the security pointer table and replaced by some other security pointer definition. In this case, a valid entry for that security pointer can be found in the security pointer table stored in the external memory. If the access is found to be consistent with some security pointer declaration, it should be allowed to continue as usual. Second, the security pointer declaration is reclaimed and no longer exists in both the SPB and the security pointer table. This represents an illegal memory reference. Execution will switch to a secure kernel handler on illegal memory reference and proper action will be taken. Two new instructions, sec_save_security_ptr and sec_load_security_ptr are proposed to support save/load security pointers in physical registers to/from the external memory. SPB and security pointer definition table are part of the protected process context. They are securely preserved during process context switch.

3.4 Integrity Protection Under MESA

The existing integrity protection schemes for security architectures are based on an m-ary hash/MACtree [7]. Under the memory-centric model in which information within a process space is usually encrypted by multiple encryption keys, the existing authentication methods cannot be directly applied. Toward this, we generalize the integrity protection tree structure. We first protect individual memory capsules with their own hash/MAC (message authentication code) tree,

and then, a hierarchical hash/MAC tree is constructed over these capsule-based hash/MAC trees. To speed up integrity verification, frequently accessed nodes of the hash/MAC trees are cached on-chip. When a new cache line is fetched, the secure processor verifies its integrity by inserting it into the MAC/hash tree. Starting from the bottom of the tree, recursively, a new MAC or hash value is computed and compared with the internal MAC/hash tree node. The MAC/hash tree is always updated whenever a dirty cache line is evicted from the secure processor. The secure processor can automatically determine the memory locations of MAC/hash tree nodes and fetch them automatically during integrity check if they are needed. Root of the MAC/hash tree is preserved securely when a process is swapped out of the processor pipeline.

4 Performance Evaluation

4.1 Simulation Framework

We used TAXI [9] as our simulation environment. TAXI includes two components, a functional open-source Pentium emulator called Bochs capable of performing full system simulation, and a cycle-based out-of-order x86 processor simulator. Bochs models the entire platform including Ethernet device, VGA monitor, and sound card to support the execution of a complete OS and its applications. We used Windows NT 4.0 SP6 as our target. Both Bochs and the simulator were modified for MESA. Proposed enhancement such as SAT, ALC, and SPB were modeled. We measure the performance under two encryption schemes, XOM/Aegis-like scheme and our improved scheme called *M-Tree*. The XOM/Aegis scheme uses block cipher (triple-DES and AES) for software encryption. In our evaluation, we selected AES as one encryption scheme. The other scheme is based on stream cipher which can be faster than block ciphers as the crypto-key stream can be pre-computed or securely speculated [5].

Since we have no access to the Windows source codes, simplification was taken to facilitate performance evaluation. Software handling of the MESA architecture was implemented as independent interrupt service routines. These routines maintain the SAT, ALC, and SPB using additional information on memory capsules provided by the application. Executing process is recognized by matching CR3 register (pointing to a unique physical address of process's page table) with process context.

To model the usage of MESA, we assume that the application software and the system software are separately protected. The system software includes all the system DLLs mapped to the application space including kernel32.dll, wsock32.dll, gdi32.dll, ddraw.dll, user32.dll, etc. The middleware libraries, if used, are also separately protected. Note that all the system libraries mentioned above are linked to the user space by the OS and they are invoked through normal DLL call. To track the data exchanged among the application, middleware DLLs, and system DLLs, dummy wrapper DLLs are implemented for the DLLs that interface with the application. These dummy DLLs are API hijackers. They can keep track of the pointers exchanged between an application and the

Table 1. Processor model parameters

Parameters	Values	Parameters	Values
Fetch/Decode width	8	Issue/Commit width	8
L1 I-Cache	DM, 8KB, 32B line	L1 D-Cache	DM, 8KB, 32B line
L2 Cache	4way, Unified, 32B line, 512KB	L1/L2 Latency	1 cycle / 8 cycles
MAC tree cache size	32KB	RM cache size	8KB
I-TLB	4-way, 256 entries	D-TLB	4-way, 256 entries
SHA256 latency	80ns / 80ns	ALC latency	1 cycle
SPB size	64, random replacement	SPB latency	1 cycle

system and update the security pointer table and the SPB accordingly. Usage of dynamic memory space such as heap and stack are traced and tagged. The assumption is that each protected code space uses its own separate encryption key to guarantee privacy of its stack and dynamically allocated memory space. Application images are modified using PE Explorer tool so that they will link with the wrapper DLL functions.

Table 1 lists the architectural parameters experimented. A default latency of 80ns for both SHA-256 and AES were used for the cryptographic engines in the simulation assuming that both units are custom designed. Seven NT applications were experimented: IE6.0, Acrobat Reader 5.0, Windows Media Player 2, Visual Studio 6.0, Winzip 8.0, MS Word, and Povray 3. The run of IE includes fetching webpages from yahoo.com using Bochs's simulated Ethernet driver. The run of Visual Studio includes compilation of Apache source code. Winzip run consists of decompressing package of Apache 2.0. Our run of Media Player includes play of a short AVI file of Olympics figure-skating. The input to Acrobat Reader is an Intel IA64 system programming document. The run includes search for all the appearance of the keyword "virtual memory". The run of MS Word consists of loading an IEEE paper template, type a short paragraph, cut/paste, and grammar checking. The run of Povray renders the default image.

4.2 Performance Evaluation

We first evaluate the performance of access control, and security pointer table. We use a 32-entry SAT for storing keys and security attributes. This is large

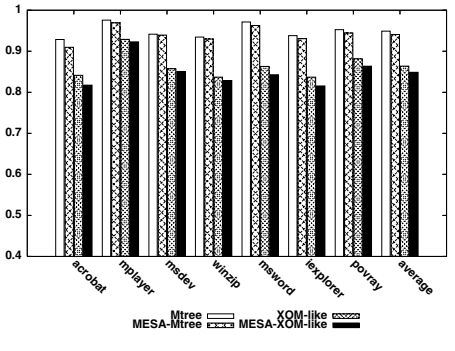


Fig. 8. Normalized IPC Results with MESA Support

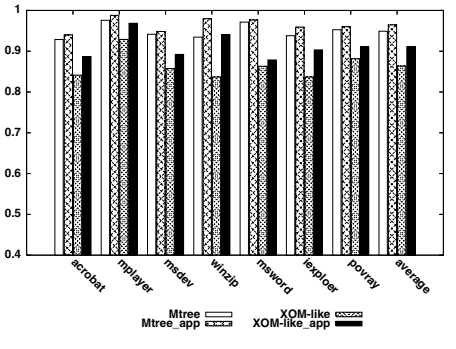


Fig. 9. Normalized IPC Results for Selective Protection

enough to hold all the protected memory capsules in our study. The number of entries in the *security pointer buffer* is 64. We evaluated three scenarios, baseline where security protection is turned off, *M-Tree* protection on the whole virtual space with one memory capsule, and protection using MESA.

Performance results for all the applications are shown in Figure 8, in which the IPC results were normalized to that of the baseline. On average, the *M-Tree* lost 5% performance. When MESA is used, it further slows down by another 1%. Block cipher on average lost 13% performance and introducing MESA will cause another 1.8% loss. Note that when the whole user space is protected as one memory capsule (degenerated case), *security pointer buffer* and access control are no longer providing any valuable service and could be turned off. There will be no further performance loss under this scenario.

One objective of MESA is to provide a secure environment so that software components co-existing in the same space could be selectively encrypted and protected. To evaluate the gain of selective protection, we selectively encrypt only application and its supporting DLLs, leave all the system libraries (e.g. gdi32.dll, wsock32.dll) un-encrypted. The normalized IPC normalized are shown in Figure 9. For *M-Tree*, the performance gain is about 1.6% comparing with encrypting everything in the memory space. This is substantial considering the slowdown of *M-Tree* is merely 5%. For the block ciphers, the gain is 5.5%. This means that by ensuring a secure environment for information sharing, MESA could actually improve the overall system performance, especially for the block cipher tamper resistant systems.

Another advantage of MESA is that it supports secure and high performance shared memory for inter-process communication. Secure inter-process communication could not be naturally supported by process-centric tamper-resistant system. The strong process isolation demands that information be either exchanged through secure socket or encrypted by software via a negotiated key between the two involved processes. MESA supports protected shared memory with almost no loss on performance. We used a micro-benchmark to evaluate the

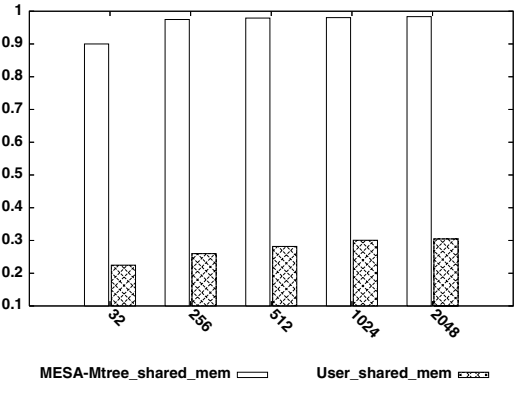


Fig. 10. Normalized Throughput of Protected Inter-process Shared Memory for MESA M-Tree under Different Shared Memory Sizes

performance of shared memory under MESA vs. a process-centric architecture. A pair of Windows producer and consumer programs using shared memory were written for this purpose. There are three scenarios, 1) the shared memory is not protected, which is used as the baseline; 2) the shared memory is protected as an encrypted secure memory capsule; 3) the shared memory is protected by applications themselves through a strong cipher (AES) with a separately negotiated key by the producer and consumer. The pair of programs were simulated in TAXI and throughput results were collected for the above three scenarios under different size of shared memory. Figure 10 clearly shows the advantage of MESA for supporting confidentiality and privacy of shared memory. For small size shared memory, the loss of throughput is about 10%. But as the size of shared memory grows, MESA protected shared memory performs almost as fast as one without protection.

5 Related Work

Software protection and trusted computing are among the most important issues in information security. Traditionally, the protections on software are provided through a trusted OS. To improve the security model, some tamper resistant devices are embedded into a computer platform to ensure the loaded OS is trusted. A typical example of such endeavor is the TPM and the related OS. Although these systems provide authentication services and prevent some simple tampering on the application, they are not designed for protection of software confidentiality and privacy against physical tampering. To address this issue, new secure processor architecture, e.g. XOM and AEGIS, emerged. However, as mostly closed system solutions, they too fail to address some very important issues such as inter-operation between heterogeneous software components and information sharing. MESA is also different from the information flow security model such as RIFLE [8]. RIFLE keeps track of the flow of sensitive information at runtime by augmenting software's binary code. It prevents restricted information from flowing to a less secure or un-trusted channel. MESA is designed for a different purpose and usage model. The main purpose of MESA is to mitigate some of the drawbacks associated with the whole process based cryptographic protection of software instead of trying to solve the issue of secure information flow. MESA is also significantly different from the memory protection model called Mondrian [10]. Memory capsules in MESA are authenticated and encrypted memory spaces, concepts do not exist in Mondrian.

6 Conclusions

This paper describes MESA, a high performance memory-centric security architecture that is able to protect software privacy and integrity against both software and physical tampering. Different from the previous process-centric tamper-resistant systems, our new system allows different software components with different security policies to inter-operate in the same memory space. It

facilitates software vendors to devise their own protections on software components therefore more flexible and suitable to open software system than the previous process-centric protection approaches.

Acknowledgment

This research was supported by NSF Grants CCF-0326396 and CNS-0325536.

References

1. E. Cohen and D. Jefferson. Protection in the hydra operating system. In *Proceedings of the 5th Symposium on Operating Systems Principles*, 1975.
2. D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectual Support For Copy and Tamper Resistant Software. In *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, 2000.
3. D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the Symposium on Operating Systems Principles*, 2003.
4. R. M. Needham and R. D. Walker. The Cambridge CAP Computer and its Protection System. In *Proceedings of the Symposium on Operating Systems Principles*, 1977.
5. W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In *Proceedings of the International Symposium on Computer Architecture*, 2005.
6. E. G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings 0f the 36th Annual International Symposium on Microarchitecture*, 2003.
7. E. G. Suh, D. Clarke, M. van Dijk, B. Gassend, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing . In *Proceedings of the International Conference on Supercomputing*, 2003.
8. N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the International Symposium on Microarchitecture*, 2004.
9. S. Vlaovic and E. S. Davidson. TAXI: Trace Analysis for X86 Interpretation. In *Proceedings of the 2002 IEEE International Conference on Computer Design*, 2002.
10. E. J. Witchel. *Mondrian Memory Protection*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2004.
11. J. Yang, Y. Zhang, and L. Gao. Fast Secure Processor for Inhibiting Software Piracty and Tampering. In *Proceedings of International Symposium on Microarchitecture*, 2003.
12. X. Zhuang, T. Zhang, and S. Pande. HIDE: an Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.
13. X. Zhuang, T. Zhang, S. Pande, and H.-H. S. Lee. HIDE: Hardware-support for Leakage-Immune Dynamic Execution. Technical Report GIT-CERCS-03-21, Geogia Institute of Technology, 2003.

A Novel Batch Rekeying Processor Architecture for Secure Multicast Key Management

Abdulhadi Shoufan, Sorin A. Huss, and Murtuza Cutleriwala

Integrated Circuits and Systems Lab, Technische Universitaet Darmstadt,
Hochschulstrasse 10, 64289 Darmstadt, Germany
{shoufan, huss, cutleriwala}@iss.tu-darmstadt.de

Abstract. The performance of secure multicast is based on an efficient group rekeying strategy which ensures that only authorized members have access to the content. In this paper a HW-solution for group rekeying is proposed. Besides the high performance guaranteed by pipelining and dynamic tree management, the Batch Rekeying Processor enables two modes of operation: real-time and batch rekeying. In addition, our solution uses a new strategy to keep the loss of QoS and access control caused by batching within system-specific limits.

1 Introduction

While multicast is an efficient solution for group communication over the Internet, it raises a key management problem when a data encryption is desired. This problem originates from the fact that the *group key* used to encrypt data is shared between many members, which demands that this key must be changed every time the group membership changes. The process of updating and distribution of a new group key is called *group rekeying*. Fig.1 represents a Pay-TV environment as an example for a multicast scenario. A video provider *VP* utilizes a video server *VS* to deliver video content encrypted with a group key k_g. A registration and authentication server *RAS* manages the group and performs group rekeying. Every registered member gets an identity key k_d (e.g., k_0 to k_3 in Fig.1) and the group key k_g. To keep backward access control, joining a new member causes the RAS to perform a group rekeying: A new group key is generated and encrypted first with the current group key and delivered to the current members, and then with the identity key of the new member and sent to this member. Disjoining a member, however, is highly inefficient. To keep forward access control the RAS has to generate a new group key and to encrypt it with each of the identity keys of the remaining members. In other words, disjoining a member from a group having n participants costs a total of $n-1$ encryptions on the server side. Obviously, a scalability problem arises. Several methods have been proposed in the literature to cope with this problem, e.g., [1] to [11]. For our purpose three solutions are especially relevant: The algorithm of logical key hierarchy LKH ([3],[4]) reduces the rekeying costs to be logarithmic to the group size. A dedicated hardware accelerator, called the Rekeying Processor [9], improves

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 169–183, 2005.

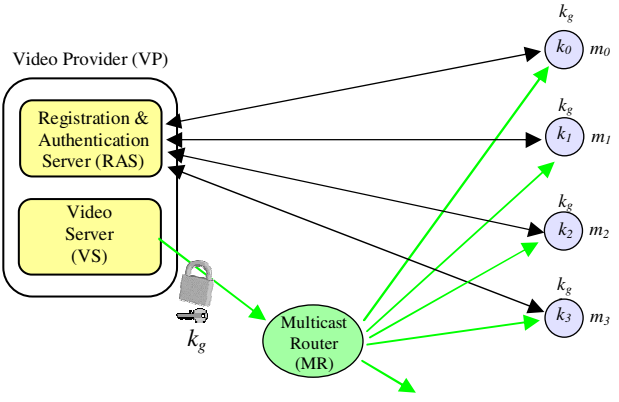


Fig. 1. Pay-TV: A potential scenario for secure multicast

the LKH performance considerably. Batch rekeying [5] can be applied to LKH to improve the rekeying efficiency, but at the cost of real-time properties. In this paper we combine these generic approaches and add new important properties which optimize the rekeying performance by means of dynamic tree management and pipelining. The proposed *Batch Rekeying Processor*, in addition, tackles the batch rekeying problems of QoS and access control and provides two operation modes: real-time and batch rekeying. Section 2 describes the preliminary work. Section 3 introduces four algorithmic improvements for group rekeying. Section 4 describes the architecture of the Batch Rekeying Processor briefly. The mapping of the algorithms on this architecture is outlined in Section 5. Section 6 details the used platform and presents some result figures. Section 7 concludes the paper with a summary and an outlook.

2 Related Work in Group Rekeying

2.1 Logical Key Hierarchy

The basic idea behind the LKH algorithm is to divide the group into hierarchical subgroups and to provide the members of each subgroup with a shared key called the *help-key* k_{x-y}. Consider the eight-member group of Fig.2. In this model members m_0 and m_1 form a subgroup with the help-key k_{0-1}. All members compose the largest subgroup with the help-key k_{0-7} which in turn corresponds to the group key used to encrypt the video data stream. For example, to disjoin member m_2 all help-keys assigned to this member have to be updated, encrypted, and sent to the remaining members. The following *rekeying message* is generated:

$$E_{k_3}(k_{2-3}^{new}), E_{k_{2-3}^{new}}(k_{0-3}^{new}), E_{k_{0-1}}(k_{0-3}^{new}), E_{k_{0-3}^{new}}(k_{0-7}^{new}), E_{k_{4-7}}(k_{0-7}^{new})$$

where the convention $E_{k_a}(k_b)$ is used to refer to a *rekeying submessage* representing the encryption of key k_b with key k_a. In contrast to the simple scheme,

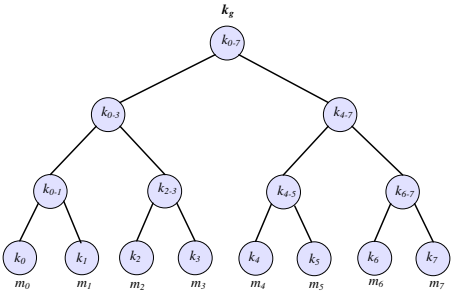


Fig. 2. LKH: Logical Key Hierarchy

where just two encryptions are needed for join rekeying, the LKH requires more encryptions. Nevertheless, considering both join and disjoin processes, the LKH is clearly superior to the simple scheme due to the logarithmic dependence on the group size.

2.2 Rekeying Processor

For performance enhancement of the LKH algorithm a hardware accelerator, called the Rekeying Processor *(RP)*, has been proposed by the authors [9]. This RP performs *real-time rekeying* and acts as a coprocessor for the registration and authentication server as shown in Fig .3. See Fig.1 for comparison. The RP handles balanced binary trees and results in a clearly higher performance compared to corresponding software solutions.

2.3 Batch Rekeying

Batch rekeying [5] proceeds in two steps which are repeated frequently. First, rekeying requests are collected and the help-keys, which need to be processed,

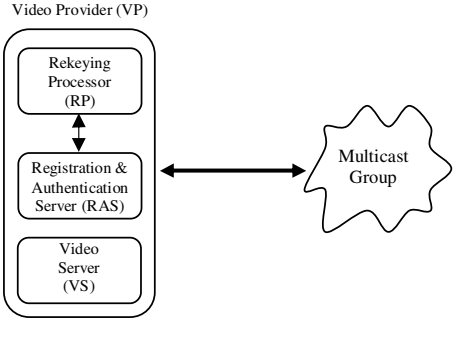


Fig. 3. RP integrated into the VP environment

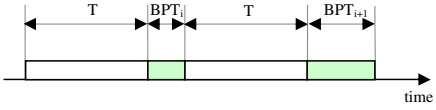


Fig. 4. Timing in batch rekeying

are marked. The marked keys build a so-called *rekeying subtree*. In a second step the rekeying subtree is processed including the generation of new keys and the building of rekeying messages. The marking is performed within regular time slots called *rekeying intervals* T. The processing takes differently long according to the current subtree. Fig.4 depicts this situation, where BPT_i denotes the *batch processing time* of the $i-th$ batch.

3 Advanced Batch Rekeying

3.1 Pipelined Batch Mode

Parallelizing the marking and the processing steps of batch rekeying results in a performace enhancement of the rekeying algorithm. During the processing of the $i-th$ batch the $(i+1)-th$ subtree can be generated as depicted in Fig.5. We define the *pipelining performance gain PPG* for one batch operation with the processing time BPT as follows.

$$PPG = (T + BPT)/T = 1 + BPT/T \tag{1}$$

3.2 Quality of Service and Access Control Aware Batch Rekeying

Batch processing of join and disjoin requests has two drawbacks with regard to Quality of Service (QoS) and access control (AC), respectively. A new member, on the one hand, gets the group key in batch rekeying later than in real-time rekeying which means that only a worse QoS can be offered by batch rekeying. On the other hand, a leaving member may exploit a valid group key in batch mode for a longer time period than in immediate rekeying which corresponds to a degradation in the AC. To quantify these items we introduce two new metrics: *LoQoS* (Loss of QoS) and *LoAC* (Loss of AC), which are defined as illustrated in Fig.6, where t_{ja}/t_{da} represents the appearance time of a join/disjoin request

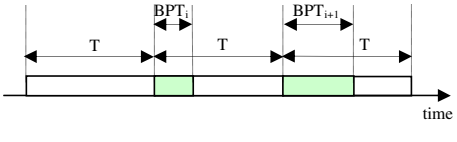


Fig. 5. Timing in pipelined batch rekeying

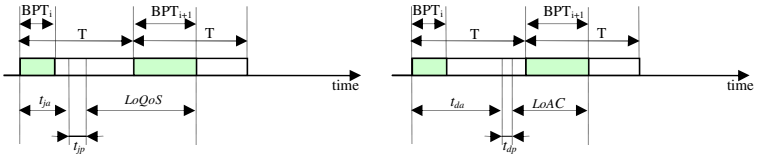


Fig. 6. LoQoS and LoAC in pipelined batch rekeying

within a rekeying interval T. t_{jp}/t_{dp} denotes the processing time of a join/disjoin request in real-time rekeying.

According to Fig.6 $LoQoS$ and $LoAC$ can be estimated as follows.

$$LoQoS = T + BPT_{i+1} - (t_{ja} + t_{jp}) \tag{2}$$

$$LoAC = T + BPT_{i+1} - (t_{da} + t_{dp}) \tag{3}$$

Batch rekeying must keep $LoQoS$ and $LoAC$ below some system-specific values denoted as $LoQoS_{max}$ and $LoAC_{max}$, respectively.

$$LoQoS < LoQoS_{max} \tag{4}$$

$$LoAC < LoAC_{max} \tag{5}$$

The conditions (4) and (5) can be best fulfilled by controlling the rekeying interval T. Accordingly, the control of $LoQoS$ and $LoAC$ results in variable rekeying intervals, which start/end at the time points, where these conditions are violated, as depicted in Fig.7. The mechanism of $LoQoS/LoAC$ control can be used to toggle between two operation modes: batch and real-time rekeying. Real-time rekeying can simply be activated by setting $LoQoS_{max}$ and $LoAC_{max}$ to zero. This causes that the conditions in (4) and (5) are violated for every join and disjoin request, which starts the processing immediately.

3.3 Dynamic Tree Management

Static tree management results in redundant operations. According to this method, the disjoin of m_2 in Fig.8 a) causes the generation of two keys and the execution of three encryptions, see Fig.8 b). The same operation costs only one generation and two encryptions, if dynamic tree management is utilized as in

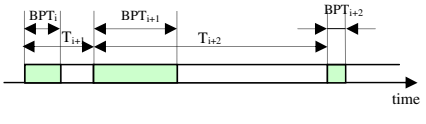


Fig. 7. LoQoS/LoAC-aware pipelined batch rekeying

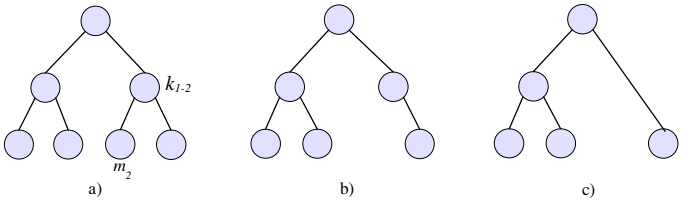


Fig. 8. Static vs. dynamic tree management

c). According to the dynamic tree management a help-key, which is not used, is deleted. The BRP, however, uses hardware-related static memory management which maps every tree node to a preassigned physical address of an SDRAM (see section 5.4). Therefore, for the BRP the concept of deletion is replaced by the concept of *suspension*. A *suspended key* is a key which is not used - but still has its place in memory - and there is no member, whose path to the root passes this key. A *right suspended* key is a key which is not used, but there is a member whose path to the root passes this key from the right. This applies for the key k_{1-2} after disjoining of m_2 in the tree c) of Fig.8. A similar definition applies for a *left suspended* key.

4 Batch Rekeying Processor

In analogy to [9] the BRP acts as a coprocessor, which communicates with the RAS over the PCI bus, according to Fig.3. The coprocessor takes the tasks of generating new help-keys, the storage of all tree keys, and the building of the rekeying messages according to the advanced batch rekeying algorithm introduced in the previous section. The BRP exploits AES-128 as an encryption algorithm [12] and the PRNG specified in ANSI X9.17 for generating new help-keys [13]. All tree keys are of length 128 bit. Each help-key, including the group key, is identified by a so-called *key identity key_ID*, which corresponds to its physical address in the SDRAM.

4.1 BRP Instruction Set

BRP executes a total of five instructions which are summarized in Table 1. The parameter *user_ID* identifies a member and corresponds to the physical address of his identity key k_d in SDRAM key memory. *Initiate_Generator* initializes the key generator according to [13]. By the instruction *Initialize_Limits* the system-specific parameters $LoQoS_{max}$ and $LoAC_{max}$ are loaded into the BRP. Finally, *Resynchronize* refreshes the help-keys of a member, if necessary.

4.2 BRP Architecture

The BRP is mainly composed of five units as depicted in Fig.9. Batch rekeying bases upon the marking of keys which need to be processed. In [5] the concept of a subtree is used to point to all marked keys in one rekeying interval. For the

Table 1. Instruction set of the Batch Rekeying Processor

Instruction	Parameter
Initiate_Generator	*D, S0,* k_{gen}
Initialize_Limits	$LoQoS_{max}$, $LoAC_{max}$
Join	*user_ID,* k_d
Disjoin	*user_ID*
Resynchronize	*user_ID*

BRP we use this concept to refer to the data generated by the Batch Scheduler to provide information on keys to be processed, the kind of processing and the order of processing. The first two information are provided by assigning a *Marking Flag (MF)* and a *Left-Right Word (LR)* to each help-key, respectively. The processing order is appointed with the aid of the *Processing FIFO (PF)*. In the sequel some functional units specific to this BRP are described briefly.

Main Controller. This subunit controls the whole BRP by fetching instructions from the Instruction FIFO, decoding them, and by activating both the Batch Scheduler to execute the marking algorithm and the Processing Unit to process an already prepared batch. The Main Controller interacts with the LoQoS/LoAC Monitor to control the rekeying interval.

Batch Scheduler. This module performs two important tasks: the marking and the estimation of batch processing time BPT. For the first task it receives a rekeying request with some $user\_ID$ and marks the help-keys needed to be processed according to the algorithm detailed in section 5.1. To estimate BPT the Batch Scheduler gets during marking the information on the number of needed key generations and encryptions. Accordingly, it updates the batch processing time BPT for each new request and delivers it to the LoQoS/LoAC Monitor.

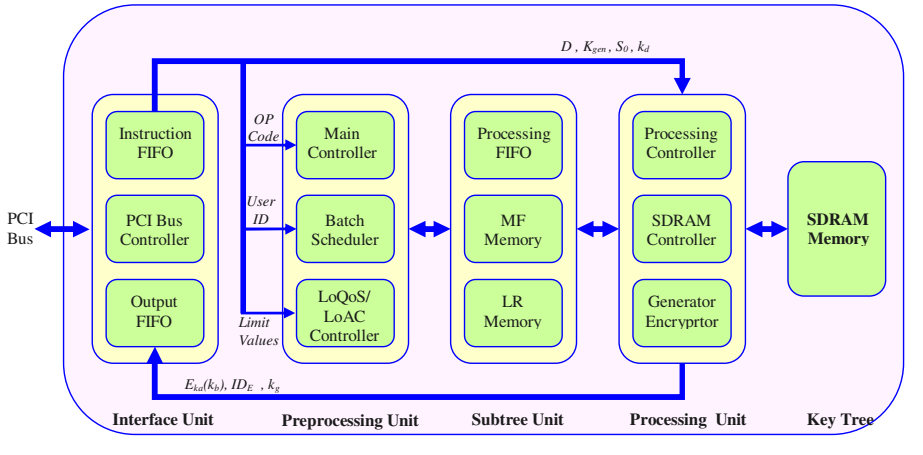


Fig. 9. Batch Rekeying Processor Architecture

Table 2. LR specification

LR	Effect in Preprocessing Unit	Effect in Processing Unit
00	*Suspended*	*No operation on this key*
01	*Right suspended*	*Update the next ancestor (whose LR=11) and encrypt it with the next right descendant (whose LR=11) or with the identity key.*
10	*Left suspended*	*Update the next ancestor (whose LR=11) and encrypt it with the next left descendant (whose LR=11) or with the identity key.*
11	*Used*	*Update and encrypt by both descendant keys*

LoQoS/LoAC Monitor. This module calculates the $LoQoS/LoAC$ in real-time and compares them with $LoQoS_{max}/LoAC_{max}$. It interrupts the Main Controller as soon as a limit is exceeded in order to start a new rekeying interval.

MF Memory. This memory saves an 1-bit marking flag for each help-key. The MF is set by the Preprocessing Unit if the corresponding key needs to be processed and is reset by the Processing Unit after the processing is completed.

LR Memory. This unit saves a 2-bit word called LR for each help-key. The LR word is used by the Batch Scheduler as a state variable of the corresponding help-key. Depending on the LR word, the Processing Unit decides on the kind of processing. Table 2 specifies how an LR word is interpreted by the Pre/Processing Units.

Processing FIFO. During marking the Batch Scheduler pushes the key_IDs of all level-1 help-keys need to be processed into this FIFO. The Processing Unit pops these key_IDs and pushes the key_IDs of the next-level help-keys need to be processed in a successive way untill all keys have been processed. This mechanism ensures that the processing of a help-key of some level can only take place after the keys of lower levels have already been handeled.

Processing Controller. This module pops the key_IDs of keys to be processed from the Processing FIFO, resets their MF-flags and orders the Generator/Encryptor to build rekeying submessages according to the corresponding LR values.

Generator/Encryptor. This unit performs both the encryption, (to set-up rekeying submessages) and the key generation based on a shared AES-128 core. A dedicated *Key FIFO* saves pre-generated keys as long as no rekeying encryption is needed.

5 Mapping of Algorithms to the BRP Architecture

5.1 Marking and Processing Algorithms

In the following the hardware batch rekeying algorithm will be illustrated assuming a sequential proceeding of the marking and processing tasks, i.e., without pipelining. Consider the left key tree of Fig.10. The current MF and LR values of its help-keys are given in the second left columns of a) and b) in Fig.11,

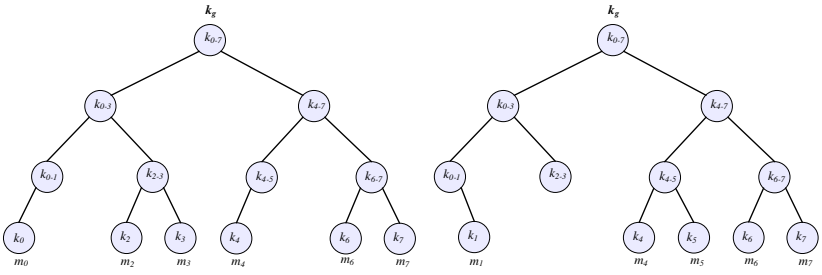


Fig. 10. Example key tree for batch rekeying

respectively. Assume that five rekeying requests occur in the current rekeying interval in the following order: Join m_1, Disjoin m_3, Disjoin m_0, Join m_5 and Disjoin m_2. The processing of these requests results in the right tree of Fig.10.

Marking. For each rekeying request the Preprocessing Unit traverses the tree from the join/disjoin point to the root and performs the following three steps:

i. The *key_ID* of the first help-key (of level-1) is pushed into the PF, if this was not yet done by other requests.
ii. All help-keys on the path are marked by setting the corresponding MF.
iii. The LR values for these keys are updated.

Fig.11 shows the development of MF, LR and PF Memory contents during the marking for the outlined example.

Processing. The generation of rekeying messages is a successive process, which is repeated as long as the PF still has entries. For each popped *key_ID* from the PF the following steps are executed.

i. Reset the corresponding MF and the MF of the father key.
ii. Push the *key_ID* of the father into the PF, if this was not yet done.
iii. Update/encrypt the corresponding help-key according to its LR value.

Fig.12 shows the development of the content of PF in the course of processing of the six help-keys in this example.

5.2 Pipelined Batch Rekeying

Pipelined batch rekeying demands simultaneous access to the Subtree Unit by the Pre/Processing Units. To avoid conflicts two MF-Memories (*Odd-MF* and *Even-MF*) and two Processing FIFOs (*Odd-PF* and *Even-PF*) are used. In contrast, for pipelining three LR-Memories are needed (*Odd-LR, Even-LR* and *Current-LR*). The access to these memories is defined in Table 3. The need of three LR memories, instead of two, can be justified as follows. The Preprocessing Unit updates an LR depending on its current value and on the request type join/disjoin. An LR is identified as current by checking the corresponding

a) MF Memory during preprocessing

Help Key	MF					
	Current Value	Join m_1	Disjoin m_3	Disjoin m_0	Join m_5	Disjoin m_2
k_{0-1}	*0*	*1*	*1*	*1*	*1*	*1*
k_{2-3}	*0*	*0*	*1*	*1*	*1*	*1*
k_{4-5}	*0*	*0*	*0*	*0*	*1*	*1*
k_{6-7}	*0*	*0*	*0*	*0*	*0*	*0*
k_{0-3}	*0*	*1*	*1*	*1*	*1*	*1*
k_{4-7}	*0*	*0*	*0*	*0*	*1*	*1*
k_{0-7}	*0*	*1*	*1*	*1*	*1*	*1*

b) LR Memory during preprocessing

Help Key	LR					
	Current Value	Join m_1	Disjoin m_3	Disjoin m_0	Join m_5	Disjoin m_2
k_{0-1}	*10*	*11*	*11*	*01*	*01*	*01*
k_{2-3}	*11*	*11*	*10*	*10*	*10*	*00*
k_{4-5}	*10*	*10*	*10*	*10*	*11*	*11*
k_{6-7}	*11*	*11*	*11*	*11*	*11*	*11*
k_{0-3}	*11*	*11*	*11*	*11*	*11*	*10*
k_{4-7}	*11*	*11*	*11*	*11*	*11*	*11*
k_{0-7}	*11*	*11*	*11*	*11*	*11*	*11*

c) Processing FIFO content during preprocessing

PF					
Current Entries	Join m_1	Disjoin m_3	Disjoin m_0	Join m_5	Disjoin m_2
				key_ID (k_{4-5})	*key_ID* (k_{4-5})
		key_ID (k_{2-3})	*key_ID* (k_{2-3})	*key_ID* (k_{2-3})	*key_ID* (k_{2-3})
	key_ID (k_{0-1})	*key_ID* (k_{0-1})	*key_ID* (k_{0-1})	*key_ID* (k_{0-1})	*key_ID* (k_{0-1})

Preprocessing Time

Fig. 11. BRP marking example

Current entries	During the processing of:					
	k_{0-1}	K_{2-3}	k_{6-7}	k_{0-3}	k_{4-7}	k_{0-7}
key_ID (k_{4-5})	*key_ID* (k_{0-3})					
key_ID (k_{2-3})	*key_ID* (k_{4-5})	*key_ID* (k_{0-3})	*key_ID* (k_{4-7})	*key_ID* (k_{0-7})		
key_ID (k_{0-1})	*key_ID* (k_{2-3})	*key_ID* (k_{4-5})	*key_ID* (k_{0-3})	*key_ID* (k_{4-7})	*key_ID* (k_{0-7})	

Processing Time

Fig. 12. Processing FIFO content during processing

Table 3. Memory operations in pipelined batch rekeying

Unit	Interval	Even MF	Odd MF	Even LR	Odd LR	Current LR	Even PF	Odd PF
Prerocessing Unit	Even	*R & W*	*No Op.*	*W*	*No Op.*	*R & W*	*W*	*No Op.*
	odd	*No Op.*	*R & W*	*No Op.*	*W*	*R & W*	*No Op.*	*W*
Processing Unit	Even	*No Op.*	*W*	*No Op.*	*R*	*No Op.*	*No Op.*	*R & W*
	odd	*W*	*No Op.*	*R*	*No Op.*	*No Op.*	*R & W*	*No Op.*

MF. Because an MF bit can be written in both odd and even intervals - see columns 3 and 4 of Table 3 - this bit can not inform definitively about the status of an LR. We solve this problem by keeping the up-to-date LR values permanently in a third memory unit which is read and written in both intervals by the Preprocessing Unit. In addition, the current LR values are inserted into the Odd-LR Memory in odd intervals and into the Even-LR Memory in even ones.

5.3 LoQoS/LoAC Monitor

Fig.13 depicts the basic architecture of the LoQoS/LoAC Monitor. Considering (2) and (3) in section 3.2, the contribution of the real-time processing times t_{jp} and t_{dp} is very small compared to rekeying interval and batch processing times and can therefore be neglected. The signals $Restart_j$ and $Restart_d$ in Fig.13 are set by the Batch Scheduler as soon as the first join/disjoin request appears in the current rekeying interval, respectively. With this measure the $LoQoS/LoAC$ can be calculated as a sum of the rekeying interval and the batch processing time.

5.4 Dynamic Tree Management

The SDRAM saves all the 128-bit keys used by the group, i.e., a total of $2n-1$ keys. Logically, the memory represents a binary tree of keys, i.e., the keys are saved on several levels. Assuming n is a power of 2, then the number of levels

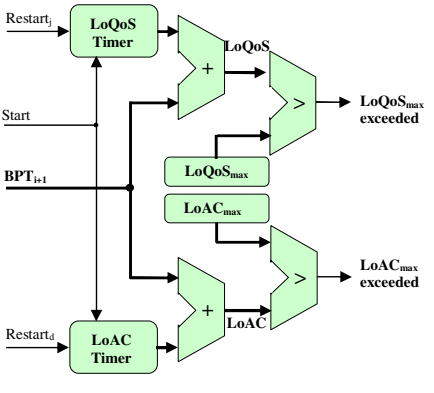


Fig. 13. LoQoS/LoAC Monitor

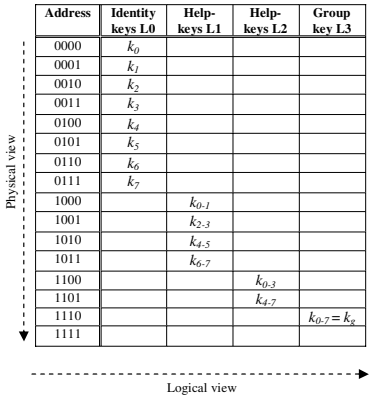


Fig. 14. Storage of the key tree in SDRAM

is determined by $L = 1 + log_2 n$. The identity keys $k'_d s$ always occupy the lowest level and the group key is located exclusively on the last level as detailed in Fig.14. Physically, the identity key of the first member is saved at the first address and the group key at the last address. This arrangement allows for an efficient traverse of the tree. An example for a group of 8 members ($n = 8, L = 4$) illustrates this approach. The group needs a total of $2n - 1 = 15$ keys, which are stored in the memory as depicted in Fig.14. Considering a key k_i in the tree, it can be seen that the addresses of the relatives of k_i can be found by performing only some shift and bit inversion operations on the address of k_i. The parent of k_0, for example, is located at the address resulting from a right shift of the address of k_0 with an insertion of 1 from left.

6 Implementation and Results

6.1 Hardware Platform and Resource Usage

As hardware platform we use the PCI card ADM-XPL from Alpha Data, Inc. [14]. The card is equipped with a VirtexII-Pro FPGA 2VP20 from Xilinx, Inc. [15]. The FPGA implements all of the BRP components except for the key memory, which is embodied using a DDR SDRAM. The Subtree Unit is implemented using the Block Select RAMs (BSRAMs) of VirtexII-Pro to enable a fast access to the MF/LR words and accordingly an early decision about the operation to be performed. Table 4 details the resource usage by the BRP obtained from exercising the synthesis tool Synplify Pro 7.3.3 [16].

6.2 Maximal Group Size N_{max} and Maximal Batch Size B_{max}

The maximal group size is limited by the available memory on the hardware card. For each help-key a total of 8 auxiliary bits on BRAMs are needed for saving MF and LR values. Corresponding to Table 4 only 64 blocks are available

Table 4. BRP resource usage

UNIT	COMPONENT	AREA USAGE	COMMENTS
Interface Unit	Instruction FIFO	4 BSRAMs 0 CLBs	Rekeying Instructions
	PCI Bus Controller	3% CLBs	Control of PCI transfers
	Output FIFO	4 BSRAMs 0 CLBs	Rekeying messages
Preprocessing Unit	Main Controller	0.26% CLBs	Control of entire BRP
	Batch Scheduler	0.69% CLBs	Marking and BPT estimation
	LoQoS/LoAC Monitor	0.86% CLBs	Control of LoQoS and LoAC
Subtree Unit	Processing FIFO	2 BSRAMs 0% CLBs	Storage of Key_IDs of help-keys to be processed
	LR & MF memory	64 BSRAMs 0% CLBs	Storage of MF and LR words
Processing Unit	Processing Controller	1.12% CLBs	Control of Processing Unit
	SDRAM Controller	2 DCM 3% CLBs	Control of SDRAM
	Generator/Encryptor	14 BSRAMs 11% CLBs	10 BSRAMs for SBOX of AES + 4 BSRAMs for key FIFO
Key Tree	SDRAM	SDRAM = 64 MB	External to the FPGA

for this task. Accordingly, the following number of help-keys can be managed by the BRP: $64 * 16Kbit/8 = 131,072$. This corresponds to a group size of $N_{max} = 131,072$ members and a level number of $L_{max} = log_2 N_{max} = 17$. The maximal batch size B_{max} is limited by the maximal number of Key_IDs that can be pushed into the PF during marking. In the current BRP prototype the PF has a capacity of 512, which allows for a maximal batch size of 1024 requests.

6.3 Performance

In its compact implementation, the AES core demands 127 clock cycles to perform one encryption. Accordingly, the generation of a new key takes 253 clock cycles. The BRP implementation using ISE 6.1 from Xilinx [15] resulted in a maximal clock frequency of about 150 MHz. The SDRAM outside the FPGA, however, can be clocked maximally at 133 MHz. The BRP, therefore, is clocked at the same frequency. To assess the performance gain of the BRP we built a SW model for batch rekeying with the same algorithms and optimization strategies of the BRP and executed it on two processors:

SW1: AMD Duron 750 MHz, 64 KB (cache), 256 MB (RAM),
SW2: Intel XEON 2.4 GHz, 512 KB (cache), 1 GB (RAM).

Table 5 depicts a performance comparison for the worst-case join/disjoin operations which occur for full trees. Assume a fixed rekeying interval of 1 second and - for simplicity - that 1000 worst-case join requests occur in a rekeying interval. The batch processing time on SW2 would then be $BPT = 390\ ms$ and on the processor $BPT = 66\ ms$. Accordingly, the hardware acceleration caused by the BRP for this batch results in 590%. This estimation is still pessimistic because

Table 5. BRP performance vs. software solutions

	SW1	SW2	HW (BRP)
Worst case single join	*1.068 ms*	*0.390 ms*	*66.08 µs*
Worst case single disjoin	*1.051 ms*	*0.387 ms*	*65.12 µs*

it does not consider the contribution of pipelining. The pipelining performance gain PPG of BRP can be calculated using (1) to: $PPG = 1 + 0.066, i.e., 106\%$. To demonstrate the effect of dynamic key management we compare the costs caused by the BRP with those caused by the preliminary rekeying processor for a first join request. Whereas the RP takes 49.85 μs, the BRP takes just 3.29 μs. The first join request represents the best case for BRP, because only the group key must be generated and encrypted by the identity key of the joining member. In the RP, in contrast, 17 help-keys have to be generated and encrypted (see section 6.2). In general, the performance gain caused by dynamic tree management decreases for an increasing tree size.

7 Summary and Outlook

We proposed a HW solution to the group rekeying problem which features mainly four novelties: pipelining of the rekeying algorithm, control of QoS and access control, enabling of two rekeying modes and dynamic tree management. In the future, the BRP will undergo several improvements. For instance, a module for data source authentication will be added to protect members from forged or false rekeying messages. Another development is the placement of MF/LR storage into the external DDR SDRAM thus releasing the BSRAMs to unroll some AES rounds. This will enhance the BRP performance considerably and will allow for the management of larger groups.

References

1. Mittra, S.: Iolus: A Framework for Scalable Secure Multicasting. In Proc. of ACM SIGCOMM, Cannes, September 1997.
2. Briscoe, B.: MARKS: Zero Side Effect Multicast Key Management Using Arbitrarily Revealed Key Sequences. In proc. of First International Workshop on Networked Group Communication (NGC), Pisa, November 1999.
3. Wong K., Gouda M., Lam S.: Secure Group Communications Using Key Graphs. IEEE/ACM Trans. On Networking, Vol. 8, No. 1, pp. 16-30, Feb. 2000.
4. Wallner D., Harder E., Agee R.: Key Management for Multicast: Issues and Architectures. RFC 2627, IETF, June 1999.
5. Li X., Yang Y. R., Gouda M., Lam S. S.: Batch Rekeying for Secure Group Communications. In Proc. of Int. WWW Conf., Hong Kong, May 2001.
6. Yang Y. R., et al.: Key Management for Large Dynamic Groups: One-Way Function Trees and Amortized Initialization. draft-irtf-smug-groupkeymgmt-oft-00.txt, August 2000, work in progress.

7. Yang Y. R., et al.: Reliable Group Rekeying: Design and Performance Analysis. In Proc. of ACM SIGCOMM, San Diego, CA, August 2001.
8. Zhang X. B., et al.: Protocol Design for Scalable and Reliable Group Rekeying. In Proc. of SPIE Conference on Scalability and Traffic Control in IP Networks, Denver, August 2001.
9. Shoufan A., Huss S. A.: A Scalable Rekeying Processor for Multicast Pay-TV on Reconfigurable Platforms. Workshop on Application Specific Processors, IEEE/ACM Int. Conf. on Hardware/Software Codesign and System Synthesis, Stockholm, September 2004.
10. Goshi, Ladner: Algorithms for Dynamic Multicast Key Distribution Trees. In Proc. of ACM Symposium on Principles of Distributed Computing, Boston, 2003.
11. Challal, et al.: SAKM: A Scalable and Adaptive Key Management Approach for Multicast Communications. ACM SIGCOMM Computer Communications Review, Vol. 34, No. 2, pp. 55-70, April 2004.
12. NIST: Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197 Nov. 26, 2001.
13. NIST: Key Management Using ANSI X9.17. Federal Information Processing Standards Publication FIPS171.
14. Alpha Data Inc.: ADM-XRC-PRO-Lite (ADM-XPL), Hardware Manual.
15. Xilinx Inc.: VirtexII-Pro Data Sheet and User Guide.
16. Synplicity Inc.: Synplify Pro 7.3.3 www.synplicity.com

Arc3D: A 3D Obfuscation Architecture

Mahadevan Gomathisankaran and Akhilesh Tyagi

Iowa State University, Ames, IA 50011, USA
{gmdev, tyagi}@iastate.edu

Abstract. In DRM domain, the adversary has complete control of the computing node - supervisory privileges along with full physical as well as architectural object observational capabilities. Thus robust obfuscation is impossible to achieve with the existing software only solutions. In this paper, we develop architecture level support for obfuscation with the help of well known cryptographic methods. The three protected dimensions of this architecture Arc3D are address sequencing, contents associated with an address, and the temporal reuse of address sequences such as loops. Such an obfuscation makes the detection of good tampering points infinitesimally likely providing tamper resistance. With the use of already known software distribution model of ABYSS and XOM, we can also ensure copy protection. This results in a complete DRM architecture to provide both copy protection and IP protection.

1 Introduction

Digital rights management (DRM) deals with intellectual property (IP) protection and unauthorized copy protection. The IP protection is typically provided through a combined strategy of software obfuscation and tamper resistance. DRM violations for software can result in either financial losses for the software developers or a loss of competitive advantage in a critical domain such as defense. Software piracy alone accounted for $31 billion annual loss [1] to the software industry in 2004.

Software digital rights management traditionally consists of watermarking, obfuscation, and tamper-resistance. All of these tasks are made difficult due to the power of adversary. Any software-only solution to achieve DRM seems to be inadequate. It leads to the classical meta-level inconsistencies encountered in classical software verification derived from Gödel's incompleteness theorem. In the end, in most scenarios, it reduces to the problem of *last mile* wherein only if some small kernel of values could be isolated from the OS (as an axiom), the entire schema can be shown to work. At this point, it is worth noting that even in the Microsoft's next generation secure computing base (NGSCB) [6], the process isolation from OS under a less severe adversary model is performed with hardware help.

The trusted computing group consisting of AMD, HP, IBM, and Intel among many others is expected to release trusted platform module (TPM) [7]. The TPM is designed to provide such a root of trust for storage, for measurement, and for reporting. It also supports establishment of a certified identity for the computing platform through its endorsement key pair. The certified identity (through public and private E_k pair) is the mechanism required for safe (trusted) distribution of the software from the vendor to the

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 184–199, 2005.

computing platform. This mechanism will be used to preserve the sanctity of the vendor provided static obfuscation. Hence, we believe that TPM provides building blocks for the proposed architecture. However, we identify additional capabilities needed to support robust 3D obfuscation. The proposed architecture obfuscation blocks can absorb TPM functionality (based on the released TPM 1.2 specifications [8]).

2 The Problem

The problems we address in this paper are as follows.

1. Associability of software to a particular CPU. (*copy protection*)
2. Verifiability of the CPU's authenticity/identity. (*copy protection, IP protection*)
3. Binary file, conforming to a standardized structure, should not reveal any IP of the software through reverse engineering. (*IP protection – static obfuscation*)
4. Any modification of the binary file should make the software unusable. (*IP protection – tamper resistance*)
5. The program execution parameters visible outside CPU should not reveal any IP of the software. (*IP protection – dynamic obfuscation*)

The first two problems are analogous to the real life problem of establishing trust between two parties followed by sharing of a secret on a secure encrypted channel. This is a well analyzed problem and solutions like Pretty Good Privacy (PGP) exist. The two unknown parties establish trust through a common trusted third party, namely, a Certification Authority (CA). The two parties then share their public keys (RSA) and hence can send messages readable only by the intended audience. This approach has been used in almost all the earlier research dealing with copy protection ([4], [2]). We will be also deploy a similar approach.

The third problem requires prevention (minimization) of information leak from the static binary file/image. This could be viewed as the problem of protecting a message in an untrustworthy channel. One possible solution is to encrypt the binary file (the solution adopted by XOM [2] and ABYSS [4]). General asymmetric/symmetric encryption is a more powerful hammer than necessary for this problem which is also less efficient computationally. An alternative approach would recognize that the binary file is a sequence of instructions and data with an underlying structure. Static obfuscation [9], [10] attempts to achieve this effect. In fact, when the white-box behavior of the obfuscated program $\mathcal{T}(\mathcal{P})$ is not differentiable from its black-box behavior, for all practical purposes $\mathcal{T}(\mathcal{P})$ provides all the protection of an encrypted version of $\mathcal{P}$. Our solution to this is to use 3D (three dimensional) obfuscation that obfuscates the address sequencing, contents, and second order address sequencing.

Fourth problem requires the binary file to be tamper resistant. This could be interpreted as any modifications to the binary file should be detectable by the hardware. Message Digest, which is a one-way hash of the message itself, solves this problem. This once again is a generic solution which is applicable to any message transaction (which does not use the special properties of binary file). We rely upon obfuscation to provide the tamper resistance in the following way. Tampering gains an advantage for

the adversary only if the properties of the tampering point (such as the specific instruction or data at that point) are known. However, obfuscation prevents the adversary from associating program points with specific desirable properties (such as all the points that have a branch, call sites, to a specific procedure or all the data values that point to a specific address). Hence most tampering points are randomly derived resulting in the disabling of the program (which we do not consider to be an advantage to the adversary in the DRM model where the adversary/end user has already purchased rights to disable the program).

The fifth problem dictates that the CPU not trust anything outside its own trusted perimeter including any software layer. The problem is simplified by the fact that CPU can halt its operations once it detects any untrustworthy behavior. The attributes of the application program execution trace space, which the CPU has to protect, can be thought of as having three dimensions, namely, instructions (content), addresses at which the instructions are stored (address sequencing), and the temporal sequence of accesses of these addresses (second order address sequencing). All these three dimensions have to be protected in order to prevent any information leakage. This holds true even for data. These three dimensions of information space are referred to as address, content and time order.

3 Earlier Research

Earlier solutions like ABYSS [4], XOM [2] and TrustNo1 [5] exist. The major shortcoming of these solutions is that they do not exploit the software specific properties. All of them use encryption and message digest as a means to make the software tamper resistant. HIDE [3] an extension of XOM, points out the fact that the address trace gives adversary power to deduce the control flow graph (CFG) even though the instructions and data themselves were encrypted. However the solution proposed by HIDE to prevent information leak through the address and memory is weak. Figure 1 illustrates the weakness of HIDE approach. Thus it takes only $\frac{N(N+1)}{2}$ comparisons to reverse-engineer the permutation, where N is the permutation size. Assuming that there are 1024 cache blocks in a page, the strength of such a permutation is less than 2^{20}. Even in the chunk mode, which performs these permutations in a group of pages, the complexity grows quadratically and hence could be easily broken.

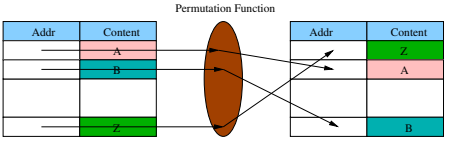


Fig. 1. Weakness of HIDE approach

4 Proposed Architecture: Arc3D

The overall proposed architecture of Arc3D is shown in Figure 3. The main affected components of the micro-architecture are the ones that handle virtual addresses. These components include the translation lookaside buffer (TLB) and page table entries (PTE). We first describe the objectives of the obfuscation schema.

4.1 Obfuscation Schema

The goal of obfuscation is to remove correlation between

1. CFG and static binary image.
2. static binary image and dynamic execution image.

Traditional static obfuscation techniques try to obscure disassembling and decompiling stages to remove correlation between the static image and CFG. But these techniques are transparent to architecture and do not remove correlation between static image and dynamic execution image. Thus an adversary monitoring the address traces could very well extract the CFG.

We use *architecture aware* obfuscation of *sequence and content* to achieve this goal. Static obfuscation, or obfuscation of the static binary image, is achieved by permuting the instruction sequence with π_S and obfuscating the instructions with C_s. Another pair of sequence permutation and content obfuscation functions which are dynamically chosen, π_D and C_D, help achieve dynamic obfuscation. These four functions, π_S,C_s,π_D and C_D form the program secret which is guarded by the trusted component of the architecture.

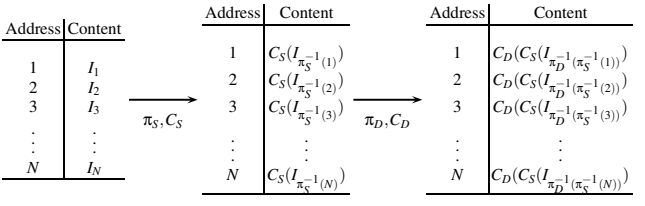


Fig. 2. Static and Dynamic obfuscation

4.2 Overall Schema

As stated earlier, Figure 3 shows the global floor-plan for the proposed architecture. The shaded areas are the additional components of *Arc3D* over the base architecture. Shading hues also indicate the access rights as follows. The lightly shaded areas contain information accessible to the outside world, *i.e.*, OS. The darkly shaded areas contain secret information accessible only to *Arc3D*. *Arc3D* has two execution modes, namely *protected* and *unprotected* mode. It has a protected register Space REG_p which is accessible only to a protected process.

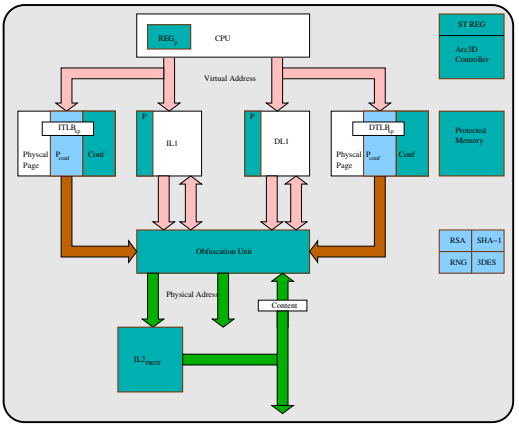


Fig. 3. Overall Schema of *Arc3D* Architecture

The core of *Arc3D* functionality is obfuscation, and it is achieved by modifying the virtual address translation path - translation look aside buffer (TLB) - of the base architecture. The TLB in addition to holding the virtual address to physical address mapping, page table entry (PTE), has the obfuscation configuration (P_{conf}). This P_{conf} is essentially the shared secrets C_S, C_D, π_S, π_D in encrypted form. In order to avoid frequent decryption, *Ar3D* stores the same in decrypted form in $Conf$ section of TLB_{xp}. This section of TLB is updated whenever a new PTE is loaded into the TLB_{xp}. *Arc3D* assumes parallel address translation paths for data and instructions, and hence Figure 3 shows DTLB and ITLB separately.

The address translation for the protected process occurs in the obfuscation unit. Sections 4.4 and 4.5 explain in detail the address sequence and content obfuscation algorithms respectively. *Arc3D* uses same logic for both static and dynamic obfuscations, and the basis of these obfuscations is the permutation function which is explained in Section 4.3. *Arc3D* has a protected L2 cache, which is accessible only to a protected process, thus providing temporal order obfuscation.

Arc3D controller provides the following interfaces (APIs) which enable the interactions of a protected process with the OS.

1. $start\_prot\_process$
2. $exit\_prot\_process$
3. $ret\_prot\_process$
4. $restore\_prot\_process$
5. $transfer\_prot\_process$

These APIs and their usage are explained in detail in Section 5.

4.3 Reconfigurable Bijective Function Unit

Obfuscation unit is a major component of *Arc3D*. This unit is responsible for generating *bijection* functions π. There are $2^n!$ possible n-bit reversible functions. Reconfigurable logic is well-suited to generate a large dynamically variable subset of these reversible functions. Figure 4 shows one such schema for permutation of 10 address bits (specifying a page consisting of 1024 cache blocks). Before explaining the blocks of Figure 4, we observe that there are $\left(2^{2^n}\right)^n$ possible functions implemented in a $n \times n$ look up table (LUT) or n n-LUTs. But only a subset of them are bijective. We wish to implement only reversible (conservative) gates ([12], [13]) with LUTs.

Both Fredkin [12] and Toffoli [14] have defined classes of reversible gates. We use *Toffoli(5,5)* gates with 5-input bits and 5-output bits in our scheme as shown in Figure 4. However, we could easily replace them by *Fredkin(5,5)* gates. The 5-input bits in a Toffoli(5,5) gate are partitioned into a control set C and a target set T. The bits in the target set are ex-ored with the bits from the control set. The exchanger blocks shown in Figure 4 perform *swap* operation. The domain of configurations mappable to each of these LUTs consists of selections of sets T and C such that $T \cap C = \emptyset$. For a support set of 5 variables, the number of unique reversible Toffoli functions is $4\binom{5}{1} + 3\binom{5}{2} + 2\binom{5}{3} + \binom{5}{4}$. Each of these terms captures control sets of size 1,2,3, and 4

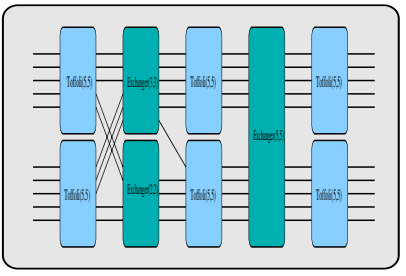


Fig. 4. Reconfigurable Bijective Obfuscation Unit

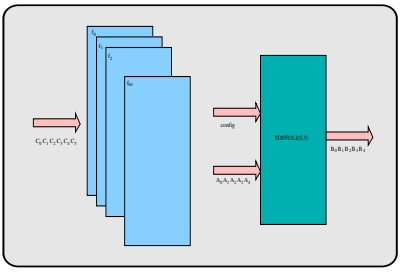


Fig. 5. Configuration Selection for each LUT

respectively. Ignoring control sets of size 1, we get a total of 55 reversible functions. Thus total permutation space covered by all six of these gates is $(55)^6 \approx 2^{34}$. There are several redundant configurations in this space. We estimated this redundancy by modeling it as as a random experiment where we have $N(=2^{39})$ balls in a basket which are either *red*(=*redundant*) or *green*(=*non-redundant*) and estimate the number of red balls in the basket by picking $n(=2^{20})$ balls. The results of our experiment show that on the average only 0.3% of the configurations are redundant with 99% probability.

Having fixed the reconfigurable logic to perform the obfuscation (permutation), we need to develop a schema for the LUT configuration. A simple mechanism would be to store all the 55 possible configurations at each of the LUTs (similar to DPGA of DeHon [15]). In addition to 4 *input bits*, each LUT will also have 6 *configuration bits* to choose one of the 55 configurations (assuming some configurations are repeated to fill the 64 locations), as shown in Figure 5. Each of the *exchanger* blocks also requires 1 configuration bit. Thus a total of 39 configuration bits are needed by the reversible logic of Figure 4.

4.4 Obfuscating the Sequence

We can use the reconfigurable unit defined in Section 4.3 to achieve sequence obfuscation. Note that even though we have shown the circuit for 10 *bits*, the methodology is applicable to an arbitrary number of address bits. We do need at least 10 address bits to provide a reasonably complex permutation space. Another reason to use 10 *bits* is due to the structure of the software. Software (both instruction and data) as we know is viewed by the architecture in various units of sizes (granularities). While residing in RAM it is viewed in units of *pages* and while residing in cache it is viewed in units of *blocks*. An obfuscation of sequences of these units is well-suited to the software structure. Hence we obfuscate the sequence of *cache blocks* within a page. We are limited to permutations within a page by the following reason as well. Page management is done by the OS and any obfuscation that crosses *page* boundary has to expose the permutation function (π) to the OS. This is also the reason why we cannot obfuscate sequences of pages. In the other direction, permuting the sequences of sub-units of cache blocks seriously affects the locality of cache resulting in severe performance degradation. Moreover, since the contents of a cache block are obfuscated, the information

leak through the preserved, original sequence of cache sub-blocks is minimized. With a *page* size of 64KB and 64B *cache blocks*, as is the case with Alpha-21264, we get 1024 *cache blocks* per page, *i.e.*, 10 *bits* of obfuscation.

4.5 Obfuscating the Contents

In cryptography, the *one time pad* (OTP), sometimes known as the *Vernam cipher*, is a theoretically unbreakable method of encryption where the plaintext is transformed (for example, XOR) with a random *pad* of the same length as the plaintext. The structured nature of software comes in handy for us once again. We can consider software as a sequence of fixed sized messages, *i.e.*, *cache blocks*. Thus if we have unique OTPs for each one of the cache blocks in the software, it is completely protected. However maintaining that many OTPs is highly inefficient. Moreover we at least have to guarantee that every *cache block* within a *page* has a unique OTP. This is to overcome the weakness related to HIDE as explained in Section 3, Figure 1. If the adversary-visible contents of the memory locations are changed after each permutation (as with unique cache block OTP per page), then n-bit permutation is $2^n!$ strong. This is in contrast with the strength of the order of 2^n exhibited by the original HIDE scheme.

In order to achieve the unique cache block OTP per page property, one option is to generate a random OTP mask for each cache block for each page. A more efficient solution, however, is to pre-generate N_b OTPs for every cache block within a page ($OTP[b_i]$ masks for $0 \leq b_i < N_b$ for a cache with N_b blocks). However, the association of an OTP with a cache block is randomized with the π_c function. The π_c function can be chosen differently for each page to provide us with the unique cache block OTP per page property. This simplifies the hardware implementation of content obfuscation unit as well since each page is processed uniformly in this unit except for the π_c function. Hence a software image will need to provide a page of OTPs which will be used for all the pages for this software. Additionally, it also needs to specify a unique mapping function π_c per page. Since we already have the reconfigurable permutation logic of Section 4.3 in *Arc3D*, we can use it to implement the function π_c as well. This results in 39 *bits* per page overhead for the specification of the content obfuscation. As with any OTP related encryption the main problem is not OTP generation, but it is OTP sharing. While estimating the complexity of the system we need to take this into consideration. Although we use XOR as our OTP function, it can be easily replaced with any other *bijective* function.

4.6 Obfuscating Temporal Order (Second Order Address Sequences)

The second order address sequence obfuscation strives to eliminate correlations between the order traces from any two iterations. Interestingly, the second order address sequence obfuscation is an inherent property of a typical computer architecture implementation. The access pattern we observe outside CPU is naturally obfuscated due to various factors like *caching*, *prefetching*, and various other *prediction* mechanisms aimed at improving the performance. But these architecture features are also controllable, directly or indirectly, by OS and other layers of software. For example, the adversary could flush the cache after every instruction execution. This renders the obfuscation effect of *cache* non-existent. To overcome such OS directed attacks, it is

sufficient to have a reasonably sized *protected cache* in the architecture which is *privileged* (accessible to secure processes alone). We expect that *page* sized, in our case 64KB, *cache* should be sufficient to mask the effects of loops. Encrypted or content-obfuscated *cache blocks* already obfuscate CFGs (within the cache block) as 64B can contain 16 instructions if we assume instructions of length 32-bits.

5 Arc3D in Operation

We have developed and described all the building blocks of *Arc3D* in Section 4. In this section, we explain its operation with respect to the software interactions in detail, from software distribution to the management of a protected process by OS using APIs provided by *Arc3D*.

5.1 Software Distribution

Arc3D provides both *tamper resistance* and *IP protection* with obfuscation. Hence, a software vendor should be able to obfuscate the static image of the binary executable. Moreover, a mechanism to distribute the de-obfuscation function configuration from the vendor to *Arc3D* needs to be supported. This configuration constitutes the shared secret between the vendor and *Arc3D*. Trust has to be established between *Arc3D* and the vendor in order to share this secret. Once the trust is established, the binary image along with the relevant configuration information can be transferred to *Arc3D*.

Trust Establishment. We assume that there exist protected elements within the CPU which are accessible only to the architecture, and not to any other entities. We also assume that every CPU has a *unique identity*, namely, its *public-private key pair* $(E_k{}^+, E_k{}^-)$. This key pair is stored in the protected space of the CPU. Public part of this key pair, $E_k{}^+$, is distributed to a *certification authority* (CA). Any party entering a transaction with the CPU (such as a software vendor) can query the CA with $E_k{}^+$ in order to establish trust in the CPU. Since CA is a well known trusted entity, the data provided by CA can also be trusted. This is very similar to the PGP model of trust establishment. An important point to note here is that trust establishment and key management mechanisms do not constitute the crux of *Arc3D* architecture. *Arc3D* could use any model/policy for this purpose. We use this model for illustration purposes only. It could very well be adapted to use the TPM [7] model.

Binary Image Generation. Software vendor generates two sets of random configuration bits per page. One configuration is to obfuscate the sequence of *cache block* addresses within a page (π_S) and the second configuration is to obfuscate the association of OTPs with *cache block* address (π_{c_s}). The content obfuscation requires the software vendor to further generate a page sized OTP ($OTP_s, OTP[b_i]$ for all $0 \leq b_i < N_b$). These functions can then be used with the FPGA obfuscation unit in a CPU or with a software simulation of its behavior to generate the obfuscated binary file. Since this obfuscation is performed on static binary images, which do not have any run-time information, we call this *static obfuscation*. The basis of static obfuscation is a page obfuscation function (*page_obfuscate*) which takes an input page, an OTP page, configuration selection bits for both address sequence and

content, and produces an output page. The outline of this algorithm is shown in Algorithm-1. The algorithm for *static obfuscation* is shown in Algorithm-2.

Software vendor generates S_{seq}, configuration selection for sequence obfuscation (corresponding to π_S), for every page to be protected and S_{cont}, configuration selection for content obfuscation (corresponding to π_{c_s}), and uses *page_obfuscate* to obfuscate the page, and associate the configuration information with the page. This is explained in Algorithm-2. Even for pages which are not loaded, obfuscation function could be associated. Note that *Arc3D* needs a standardized mechanism to garner these functions. This could be done by extending the standard binary format, like ELF, to hold sections containing the configuration information. These configuration bits have to be guarded, and hence need to be encrypted before being stored with the binary image. Thus a key K_s is generated and used to encrypt this new section. HMAC [16], which is a keyed hash, is also generated for this new section in order to prevent any tampering.

Algorithm 1. Page Obfuscation Function: *page_obfuscate*

Required Functions
$F_{obf}(conf\_sel, addr) \Leftarrow$ Reconfigurable Obfuscation Unit
Inputs
$OTP_{arr} \Leftarrow$ array of OTP
$page_i \Leftarrow$ input page
$conf_{seq} \Leftarrow$ conf_sel for sequence obfuscation
$conf_{cont} \Leftarrow$ conf_sel for content obfuscation
$N_b \Leftarrow$ number of *cache blocks* in a page
Outputs
$page_o \Leftarrow$ output of page
Function
for $k = 0$ to $N_b - 1$ **do**
  $out = F_{obf}(conf_{seq}, k)$
  $l = F_{obf}(conf_{cont}, k)$
  $OTP = OTP_{arr}[l]$
  $page_o[out] = page_i[k] \oplus OTP$
end for

Algorithm 2. Static Obfuscation Function: *stat_obfuscate*

Inputs
$N_p \Leftarrow$ number of pages in the binary
$Page_{arr} \Leftarrow$ array of pages
Function
$p \Leftarrow$ temporary page
Generate random page of OTP (OTP_s)
for $k = 0$ to $N_p - 1$ **do**
  if $Page_{arr}[k]$ to be protected **then**
    Generate random S_{seq}
    Generate random S_{cont}
    $Page_{arr}[k].p_{conf} = \{K_s\{S_{seq}, S_{cont}\}, HMAC\}$
    $p = page\_obfuscate(S_{seq}, S_{cont}, OTP_s, Page_{arr}[k].page)$
    $Page_{arr}[k].page = p$
  end if
end for

In order for the CPU to be able to decrypt the program, it needs the key K_s. This is achieved by encrypting K_s with ${E_k}^+$ and sending it with the software to the CPU. Now only the CPU with the private key ${E_k}^-$ can decrypt the distributed image to extract K_s. The entry point of the software also needs to be guarded. Several attacks are possible if the adversary could change the entry point. Hence, the entry point is also encrypted with K_s. Once again we need to use HMAC to detect any tampering. Hence, S_{auth}, the authorization section, consists of ${E_k}^+\{K_s, PC_{start}\}, HMAC$. The complete algorithm for software distribution step is shown in Algorithm-3.

5.2 Management of Protected Process

In this section we will explain, in detail, how OS uses the API provided by *Arc3D* controller to manage a protected process. We will also see how seamlessly it can be integrated with the existing systems while providing the guarantees of *tamper resistance* and *copy protection*.

Starting a Protected Process: *Arc3D* has two execution modes, (1) protected and (2) normal, which are enforced without necessarily requiring the OS cooperation. When the OS creates a process corresponding to a protected software, it has to read the special sections containing S_{auth} and per-page configuration P_{conf}. *Arc3D* has an extended translation lookaside buffer (TLB_{xp}) in order to load these per-page configuration bits. The decision whether to extend page table entry (PTE) with these configuration bits is OS and architecture dependent. We consider an architecture in which TLB misses are handled by the software and hence OS could maintain these associations in a data structure different from PTEs. This will be efficient if very few protected processes (and hence protected pages) exist. This method is equally well applicable to a hardware managed TLB wherein all the PTEs have to follow the same structure.

The OS, before starting the process, has to update extended TLB with P_{conf}, for each protected page (a page which has been obfuscated). Additionally, for every protected page, OS has to set the protected mode bit P. This will be used by the architecture to decide whether to use obfuscation function or not. Note that by entrusting the OS to set the P bit, we have not compromised any security. The OS does not gain any information or advantage by misrepresenting the P bit. For example, by misrepresenting a protected page as unprotected, the execution sequence will fail as both instructions and address sequences will appear to be corrupted. This is followed by the OS providing *Arc3D* with a pointer to S_{auth} and a pointer to S_{OTP}.

Algorithm 3. Software Distribution

1: Get $E_k{}^+$ from CPU
2: Contact CA and validate $E_k{}^+$
3: Generate K_s
4: Generate $conf\_seq, conf\_cont$ for every page to be protected
5: Generate OTP page
6: Do $stat\_obfuscate$
7: Generate S_{auth} and add it to binary file
8: Generate S_{conf} and add it to binary file
9: Generate S_{OTP} and add it to binary file
10: Send the binary file to CPU

Algorithm 4. Dynamic Obfuscation Function: $dyn\_obfuscate$

Inputs
$N_{TLB} \Leftarrow$ number of TLB entries
$p_i \Leftarrow$ page to be obfuscated, read from RAM
$p_o \Leftarrow$ obfuscated page
$OTP_d \Leftarrow$ array of dynamic OTP
Function
for $k = 0$ to $N_{TLB} - 1$ **do**
  if $TLB[k].P$ is set **then**
    if $TLB[k].prot = NULL$ **then**
      Decrypt and validate P_{conf}
      if D_{seq}, D_{cont} exist **then**
        $p_o = page\_unobfuscate(D_{seq}, D_{cont}, OTP_d, temp_i)$
        $p_i = temp_o$
      end if
      Generate new D_{seq}, D_{cont}
      Append it to P_{conf}
      $TLB[k].prot = \{S_{seq}, S_{cont}, D_{seq}, D_{cont}\}$
      Read the page in p_i
      $p_o = page\_obfuscate(D_{seq}, D_{cont}, OTP_d, p_i)$
      Write back $temp_o$
    end if
  end if
end for

The OS executes $start\_prot\_process$ primitive to start the protected process execution. This causes *Arc3D* to transition to *protected* mode. *Arc3D* decrypts S_{auth} and checks its validity by generating the HMAC. If there is any mismatch between the computed and stored HMACs, it raises an exception and goes out of *protected*

mode. If HMACs match, then *Arc3D* can start the process execution from PC_{start}. Once *start_prot_process* is executed, *Arc3D* generates an OTP page (OTP_d). This OTP_d needs to be stored in memory so that it can be reloaded at a later point after a context switch. We use the section S_{OTP} to store OTP_d. *Arc3D* has sufficient internal space to hold OTP_s and OTP_d at the same point in time. It reads S_{OTP} and decrypts OTP_s, and validates the HMAC and then loads it into the obfuscation engine. It then encrypts OTP_d with K_s and generates its HMAC which is appended to S_{OTP}. We assume that S_{OTP} has reserved space for OTP_d and its HMAC in advance.

Arc3D then scans the TLB. For every protected page that has been loaded in main memory (RAM), it validates P_{conf}. It then generates D_{seq} and D_{cont} configuration bits (corresponding to π_D and π_{c_d}) for each one of those pages and appends them to P_{conf}. TLB_{xp} which has been extended to have P_{conf}, also has protected space per TLB entry which only *Arc3D* can access. This space will be used by *Arc3D* to store the decrypted $S_{seq}, S_{cont}, D_{seq}, D_{cont}$ configuration bits, so that decryption need not be done for every TLB access. *Arc3D* contains temporary buffer of twice the *page* size to perform the obfuscation. Hence it reads a complete page from RAM and applies $page\_obfuscation$ and then stores it back in RAM. Algorithm for dynamic obfuscation is shown in Algorithm-4.

The $TLB[k].Prot$ structure is the protected section of TLB entry and is cleared every time a new TLB entry is written. Hence the function $dyn\_obfuscate$ is invoked on every TLB miss. If the page has already been subjected to dynamic obfuscation, it first performs the inverse operation (deobfuscation). It then generates new obfuscation configurations to perform dynamic obfuscation. This causes the dynamic obfuscation functions to be very short lived, *i.e.*, changing on every page fault. It makes reverse engineering of π_D and C_D functions extremely unlikely. To ensure such a (π_D, C_D) refresh on every context switch, $TLB[k].Prot$ is cleared for all the entries whenever *start_prot_process* is called or a protected process is restored. A state register ST_i is allocated to the process and added to S_{auth}. The usage of this register is explained in 5.2. Availability of this register puts a limit on total number of protected processes active at any point in time in *Arc3D*. After the dynamic obfuscation is done, the process is started from PC_{start} as given by S_{auth}. The high level steps involved in *start_prot_process* are shown in Algorithm-5.

Algorithm 5. *start_prot_process*

1: Change to *protected* mode
2: Read S_{auth} and validate
3: Read S_{OTP} and validate
4: Generate OTP_d and append to S_{OTP}
5: Clear $TLB[i].prot$ for all i
6: Call $dyn\_obfuscate$
7: Allocate ST_i to the process and add it to S_{auth}
8: Set PC to PC_{start}

Algorithm 6. TLB_{xp} Access Function: *tlbxp_access*

$v\_page \Leftarrow$ input virtual page address
$v\_block \Leftarrow$ input virtual block address
$p\_addr \Leftarrow$ output physical address
$k \leftarrow$ TLB index of hit and page exists in RAM
if $TLB[k].P$ is set **then**
  $p\_block = F_{obf}(D_{seq}, F_{obf}(S_{seq}, v\_block))$
else
  $p\_block = v\_block$
end if
$p\_addr = TLB[k].p\_page + p\_block$

Memory Access: Once a process is started it generates a sequence of instruction and data addresses. Like any high performance architecture, we assume separate TLBs for instruction and data. Hence the loading process explained earlier occurs simultaneously in both ITLB and DTLB. The TLB is the key component of the obfuscation unit. The obfuscation functions are applied only during virtual to physical memory mapping. The address generation logic is explained in Algorithm-6. Two stages of F_{obf} are in the computation path for the physical address. This makes TLB latency higher than the single cycle latency of a typical TLB access. Hence, L1 caches of both instruction and data are made *virtually tagged* and *virtually addressed* to reduce the performance impact due to TLB latency. The L1 cache tags are extended with a *protection* bit, which is accessible only to *Arc3D*. This bit is set, whenever the cache line is filled with data from a protected page. The access to cache blocks with protected bit set is restricted only in protected mode. In order to have efficient context switching mechanism we use a $write-through$ L1 cache. Thus, at any point in time L2 and L1 are in sync.

TLB and L1 cache are accessed parallely. TLB is read in two stages. The first stage reads the normal portion of TLB and the second stage reads the extended and protected portion of TLB. This way the second stage access can be direct mapped and hence could be energy-efficient. If L1 access is a hit, then TLB access is stopped at $stage_1$. If L1 access is a miss, then TLB access proceeds as explained in the function *tlbxp_access*. In *Arc3D* L2 cache is *physically tagged and physically addressed*. Hence, no special protection is needed for the L2 cache. One point to note here is that the physical addresses generated by TLB_{xp} are cache block boundary aligned (integer multiple of cache block size, 64B in our example and hence with 6 least significant 0s). This is because, as we know, there could be many instructions within a cache block, and the instruction sequence within the cache block is not obfuscated. Hence, an exposure of the actual addresses falling within a cache block leaks information. This leads to the cache block aligned address restriction. This would increase the latency but as we discuss later, this increase will not be very high. Once the data is received from the L2 cache or memory, it is XORed with both OTP_d and OTP_s to get the actual content in plaintext which is then stored in an L1 cache line.

Execution: *Arc3D* has a set of protected registers (REG_p) to support protected process execution. This register set is accessible only in the protected mode. The protected process can use the normal registers to communicate with OS and other unprotected applications. If two protected processes need to communicate in a *secure* way, then they have to use elaborate protocols to establish common obfuscation functions. Since K_s is known to the application itself, it can modify the P_{conf} such that it shares the same configuration functions with the other processes (if need be).

Interrupt Handling: Only instructions from a protected page can be executed in protected mode. Hence any services, such as dynamic linked libraries, require a state change. Any interrupt causes the *Arc3D* to go out of protected mode. Before transitioning to normal mode, *Arc3D* updates PC field in S_{auth} with the current PC. Thus a protected process context could be suspended in the background while the interrupt handler is running in the unprotected mode. When the interrupt handler is done, it can execute *ret_prot_process* to return to the protected process. *Arc3D* takes the PC from

S_{auth} and restarts from that point. This allows for efficient interrupt handling. But from the interrupt handler, the OS could start other unprotected processes. This way *Arc3D* does not have any overhead in a context switch from protected to unprotected processes. But when OS wants to load another protected process the current protected process' context must be saved.

Saving and Restoring Protected Context: *Arc3D* exports *save_prot_process* primitive to save the current protected process context. This causes *Arc3D* to write $K_s\{REG_p\} + HMAC$ and S_{auth} into the memory given by the OS. The OS when restoring the *protected* process, should provide pointers to these data structures through *restore_prot_process*. But this model has a flaw, as we don't have any association of time with respect to the saved context. Hence *Arc3D* cannot detect if there is a *replay* attack. *Arc3D* needs to keep some state which associates the program contexts with a notion of time. A set of OTP registers called state OTP registers are required within *Arc3D* for this purpose. These registers will have the same size as K_s. The number of these registers depends on how many protected processes need to be supported simultaneously. The *start_prot_process* allocates a state OTP register, ST_i, for this protected process. This association index ST_i is also stored within S_{auth}. Each instance of *save_prot_process* generates a state OTP value $OTP[ST_i]$ which is stored in ST_i state OTP register. The saved context is encrypted with the key given by the XOR of K_s and $OTP[ST_i]$. On the other hand, an instantiation of *restore_prot_process* first garners ST_i and K_s from S_{auth}. The key $OTP[ST_i] \oplus K_s$ is used to decrypt the restored context. This mechanism is very similar to the one used in all the earlier research such as ABYSS and XOM.

Supporting Fork: In order to fork a protected process, the OS has to invoke *transfer_prot_process* API of *Arc3D*. This causes a new ST_i to be allocated to the forked child process and then makes a copy of process context as *save_prot_process*. Thus the parent and child processes could be differentiated by *Arc3D*. OS has to make a copy of S_{OTP} for the child process.

Exiting a Protected Process: When a protected process finishes execution, OS has to invoke *exit_prot_process* API to relinquish the ST_i (state OTP register) allocated to the process currently in context. This is the only resource that limits the number of protected processes allowed in the *Arc3D* system. Hence *Arc3D* is susceptible to denial-of-service (DOS) kind of attacks.

Protected Cache: *Arc3D* has a protected direct mapped L2 cache of page size, i.e., $64KB$. This protected cache is used to obfuscate the second order address sequences only for instructions, as temporal order doesn't have any meaning with respect to data. Whenever there is an IL1 miss in protected mode, *Arc3D* sends request to $L2_{prot}$. Since $L2_{prot}$ is on-chip the access latency will be small. We assume it to be 1 cycle. If there is a miss in $L2_{prot}$ then L2 is accessed. $L2_{prot}$ is also invalidated whenever a protected process is started or restored.

6 Attack Scenarios

In this section we argue that Arc3D achieves our initial goals, namely, *copy protection*, *tamper resistance* and *IP protection*. Several attacks causing information leak in various dimensions could be combined to achieve the adversary's goal. These attacks could be classified into two categories — attacks that target *Arc3D* to manipulate its control or reveal its secrets. If the adversary is successful in either getting the stored secret (E_k^-) or in changing the control logic, the security assurances built upon *Arc3D* could be breached. But these types of attacks have to be based on *hardware*, as there are no software control handles into *Arc3D*. There are several possible hardware attacks, like Power Profile Analysis attacks, Electro magnetic signal attacks. The scope of this paper is not to provide solutions to these attacks. Hence we assume that *Arc3D* is designed with resistance to these hardware attacks.

The second type of attacks are white-box attacks. Such an attack tries to modify the interfaces of *Arc3D* to the external world, to modify the control. The guarantees that are provided by *Arc3D* to software in protected mode of execution are 3D obfuscation for protected pages, and unique identity per CPU. Protected mode of execution guarantees that the control is not transferred to any unauthorized code (which is undetected). *Arc3D* will fault when an instruction from an unprotected page or from a page that was protected with different K_s is fetched in protected mode. This will prevent buffer overflow kind of attacks. 3D obfuscation provides us both IP protection and tamper resistance. IP protection is achieved because at every stage of its life, the binary is made to look different, hence reducing the correlation based information leaks to the maximum extent possible.

Tampering could be performed by many means. But all of them have to modify the image of the process. Since every cache-block in every protected page potentially could have a different OTP the probability that the adversary could insert a valid content is extremely small. Applications can obfuscate new pages that are created at run-time by designating them as protected. Applications can further maintain some form of Message Digest for sensitive data, because obfuscation only makes it harder to make any educated guess, while random modification of data is still possible. In the case of instructions the probability that a random guess would form a *valid* instruction at a valid program point is extremely small.

Another form of tampering, splicing attack, uses valid cipher texts from different locations. This attack is not possible because every *cache block* in every *page* has a unique OTP and every *page* has a unique address obfuscation function. This makes it hard for the adversary to find two *cache blocks* with the same OTP. Another common attack is replay attack, where valid cipher text of a different instance of the same application is passed (replayed). As we discussed earlier, this attack is prevented by XORing K_s with a randomly generated OTP which is kept in the Arc3D state. This value is used as a key to encrypt the *protected* process's context. Thus when restoring a protected context, *Arc3D* makes sure that both S_{auth} and saved context are from the same run.

When the adversary knows the internals of the underlying architecture, another form of attack is possible. This form of attack is denial of resources, which are essential for the functioning of the underlying architecture. For example, XOM maintains a session table and has to store a *mutating register* value per session-id. This mutating register is

used to prevent any replay attacks. This kind of architecture has an inherent limitation on the number of processes it can support, *i.e.*, the scalability issue. Thus an attacker could exhaust these resources and make the architecture non-functional. This kind of attack is possible in *Arc3D* as well on the state OTP register file. We could let the context save and restore be embedded in the storage root of trust in a TPM like model. Such a model will allow Arc3D to perform in a stateless fashion which can prevent the resource exhaustion attacks.

7 Performance Analysis

We used Simplescalar [20] Alpha simulator, with memory hierarchy as shown in Figure 3, and Spec200 [19] benchmarks to do the performance analysis. Due to lack of space we have not presented the complete analysis results. The performance impact is less than 1% for most of the benchmarks.

8 Conclusion

We propose a minimal architecture, *Arc3D*, to support efficient obfuscation of both static binary file system image and dynamic execution traces. This obfuscation covers three aspects: address sequences, contents, and second order address sequences (patterns in address sequences exercised by the first level of loops). We describe the obfuscation algorithm and schema, its hardware needs. We also discuss the robustness provided by the proposed obfuscation schema.

A reliable method for obfuscation keys distribution is needed in our system. The same method can be used for safe and authenticated software distribution to provide copy protection. A robust obfuscation also prevents tampering by rejecting a tampered instruction at an adversary desired program point with an extremely high probability. Hence obfuscation and derivative tamper resistance provide IP protection. Consequently, *Arc3D* offers complete architecture support for copy protection and IP protection, the two key ingredients of software DRM.

References

1. Business Software Alliance, *Second Annual BSA and IDC Global Software Piracy Study*, Trends in software piracy 1994-2004, May 2005.
2. D. Lie et at. *Architectural support for copy and tamper resistant software*, In Proceedings of ASPLOS 2000, pages 168-177.
3. X. Zhuang et al. *HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus*, In Proceedings of ASPLOS 2004.
4. S. R. White and Liam Comerford. *ABYSS: An Architecture for Software Protection*, IEEE Transactions on Software Engineering, Vol. 16, No. 6, June 1990, pages 619-629.
5. M. Kuhn. *The TrustNo1 Cryptoprocessor Concept*, Technical Report, Purdue University, 1997-04-30.
6. Microsoft. Next-generation secure computing base, 2003.
7. Trusted Computing Platform Alliance. Trusted Platform Module, 2003.

8. TPM Design Principles, Version 1.2. Trusted Platform Module, October 2003.
9. D. Aucsmith. "Tamper Resistant Software: An Implementation"¨*Proceedings of the First International Workshop on Information Hiding*, 1996.
10. C. Collberg et al. Watermarking, tamper-proofing, and obfuscation - tools for software protection, IEEE Transactions on Software Engineering, Vol. 28, Number 8, 2002.
11. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
12. E. Fredkin and T. Toffoli. *Conservative Logic*, In International Journal of Theoretical Physics, 21(3/4), April 1982.
13. R. Bennett and R. Landauer. *Fundamental Physical Limits of Computation*, Scientific American, pages 48-58, July 1985.
14. T. Toffoli. *Reversible Computing*. Technical Report MIT/LCS/TM151/1980, MIT Laboratory for Computer Science, 1980.
15. A. DeHon. *DPGA-coupled microprocessor: Commodity ICs for the early 21st centry*, In Proc. of IEEE workshop on FPGAs for Custom Computing Machines, pages 31-39, April 1994.
16. HMAC. Internet RFC 2104, February 1997
17. Z. Cvetanovic et al. *Performance analysis of the Alpha 21264-Based Compaq ES40 System*, In Proc. of ISCA, pages 192-202, 2000.
18. Intel 80200 Processor based on Intel XSCALE Microarchitecture Datasheet, Intel, January 2003.
19. Specbench. Spec 2000 Benchmarks
20. D. Burger and Todd M. Austin. *The SimpleScalar Tool Set, Version 2.0. Computer Sciences Department Technical report #1342*. University of Wisconsin-Madison. June 1997.

Dynamic Code Region (DCR) Based Program Phase Tracking and Prediction for Dynamic Optimizations

Jinpyo Kim[1], Sreekumar V. Kodakara[2], Wei-Chung Hsu[1], David J. Lilja[2], and Pen-Chung Yew[1]

[1] Department of Computer Science and Engineering
{jinpyo, hsu, yew}@cs.umn.edu,
[2] Department of Electrical and Computer Engineering
{sreek, lilja}@ece.umn.edu
[3] University of Minnesota-Twin Cities, Minneapolis, MN 55455, USA

Abstract. Detecting and predicting a program's execution phases are crucial to dynamic optimizations and dynamically adaptable systems. This paper shows that a phase can be associated with dynamic code regions embedded in loops and procedures which are primary targets of compiler optimizations. This paper proposes a new phase tracking hardware, especially for dynamic optimizations, that effectively identifies and accurately predicts program phases by exploiting program control flow information. Our proposed phase tracking hardware uses a simple stack and a phase signature table to track the change of phase signature between dynamic code regions.

Several design parameters of our proposed phase tracking hardware are evaluated on 10 SPEC CPU2000 benchmarks. Our proposed phase tracking hardware effectively identifies a phase at a given granularity. It correctly predicts the next program phase for 84.9% of times with a comparable small performance variance within the same phase. A longer phase length and higher phase prediction accuracy together with a reasonably small performance variance are essential to build more efficient dynamic profiling and optimization systems.

1 Introduction

Understanding and predicting a program's execution phase is crucial to dynamic optimizations and dynamically adaptable systems. Accurate classification of program behavior creates many optimization opportunities for adaptive reconfigurable microarchitectures, dynamic optimization systems, efficient power management, and accelerated architecture simulation [1,2,3,6,7,12,20,21].

Dynamically adaptable systems [1,2,8,24] have exploited phase behavior of programs in order to adaptively reconfigure microarchitecture such as the cache size. A dynamic optimization system optimizes the program binary at runtime using code transformations to increase program execution efficiency. Dynamic binary translation also falls in this category. In such systems, program phase

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 203–217, 2005.

behavior has been exploited for dynamic profiling and code cache management [7,12,13,14,17,18]. For example, the performance of code cache management relies on how well the change in instruction working set is tracked by the system.

Dynamic optimization systems continuously track the program phase change either by sampling the performance counters or by instrumenting the code. The sampling overhead usually dominates in the sampling based profiling. While using a low sampling rate can avoid high profiling overhead, it may result in an unstable system where reproducibility is compromised and also may miss optimization opportunities. Using program phase detection and prediction may more efficiently control the profiling overhead by adjusting sampling rate adaptively or by applying burst instrumentation. For example, if the program execution is in a stable phase, profiling overhead can be minimized. (e.g., by lowering the sampling rate), while a new phase would require burst profiling.

A phase detection technique developed for dynamically adaptation systems is less applicable for a dynamic optimization system. This is because collecting performance characteristics at regular time intervals with arbitrary boundaries in an instruction stream is not as useful as gathering performance profiles of instruction streams that are aligned with program control structures. We introduce the concept of Dynamic Code Region (DCR), and use it to model program execution phase. A DCR is a node and all its child nodes in the extended calling context tree (ECCT) of the program. ECCT as an extension of the calling context tree (CCT) proposed by Ammon et al [23] with the addition of loop nodes. A DCR corresponds to a sub tree in ECCT. DCR and ECCT will be explained in later sections.

In previous work [1,2,3,5], a phase is defined as a set of intervals within a program's execution that have similar behavior and performance characteristics, regardless of their temporal adjacency. The execution of a program was divided into equally sized non-overlapping intervals. An interval is a contiguous portion (i.e., a time slice) of the execution of a program. In this paper, we introduce dynamic intervals that are variable-length continuous intervals aligned with dynamic code regions and exhibit distinct program phase behavior.

In traditional compiler analysis, interval analysis is used to identify regions in the control flow graph [19]. We define dynamic interval as instruction stream aligned with code regions discovered during runtime. Dynamic intervals as a program phase can be easily identified by tracking dynamic code regions. We track higher-level control structures such as loops and the procedure calls during the execution of the program. By tracking higher-level code structures, we were able to effectively detect the change in dynamic code region, and hence, the phase changes in a program execution. This is because, intuitively, programs exhibit different phase behaviors as the result of control transfer through procedures, nested loop structures and recursive functions. In [10], it was reported that tracking loops and procedures yields comparable phase tracking accuracy to the Basic Block Vector (BBV) method [2,5], which supports our observation.

In this paper, we propose a dynamic code region (DCR) based phase tracking hardware for dynamic optimization systems. We track the code signature of

procedure calls and loops using a special hardware stack, and compare against previously seen code signatures to identify dynamic code regions. We show that the detected dynamic code regions correlate well with the observed phases in the program execution.

The primary contributions of our paper are:

- We showed that dynamic intervals that correspond to dynamic code regions that are aligned with the boundaries of procedure calls and loops can accurately represent program behavior.
- We proposed a new phase tracking hardware that consists of a simple stack and a phase signature table. Comparing with existing proposed schemes, this structure can detects a small number and longer phases. Using this structure can also give more accurate prediction of the next execution phase.

The rest of this paper is organized as follows. Section 2 described related work and the motivation of our work. Section 3 describes why dynamic code regions can be used for tracking change of coarse grained phase changes. Section 4 describes our proposed phase classification and prediction hardware. Section 5 evaluates our proposed scheme. Section 6 discusses the results and compares them to other schemes. Finally, Section 7 summarizes the work.

2 Background

Phase Detection and Prediction. In previous work [1,2,3,4,7], researchers have studied phase behavior to dynamically reconfigure microarchitecture and re-optimize binaries. In order to detect the change of program behavior, metrics representing program runtime characteristics were collected. If the difference of metrics, or code signature, between two intervals exceeds a given threshold, phase change is detected. The stability of a phase can be determined by using performance metrics (such as CPI, cache misses, and branch misprediction) [3,7], similarity of code execution profiles (such as instruction working set, basic block vector) [1,2] and data access locality (such as data reuse distance) [4] and indirect metrics (such as Entropy) [6].

Our work uses calling context as a signature to distinguish distinct phases in an extended calling context tree (ECCT). Similar extension of CCT was used for locating reconfiguration points to reduce CPU power consumption [24,25], where the calling context was analyzed on the instrumented profiles. Our phase tracking hardware could find similar reconfiguration points discovered in the analysis of instrumented profiles because we also track similar program calling contexts. This is useful for phase aware power management in embedded processors. M. Huang et al [8] proposes to track calling context by using a hardware stack for microarchitecture adaptation in order to reduce processor power consumption. W. Liu and M. Huang [27] propose to exploit program repetition to accelerate detailed microarchitecture simulation by examining procedure calls and loops in the simulated instruction streams. M. Hind et al. [22] identified two major parameters (granularity and similarity) that capture the essence of phase shift detection problems.

Exploiting Phase Behavior for Dynamic Optimization System. In dynamic optimization systems [12,17,18,20], it is important to maximize the amount of time spent in the code cache because trace regeneration overhead is relatively high and may offset performance gains from optimized traces [13]. Dynamo [12] used preemptive flushing policy for code cache management, which detected a program phase change and flushed the entire code cache. This policy performs more effective than a policy that simply flushes the entire code cache when it is full. Accurate phase change detection would enable more efficient code cache management. ADORE [7,20] used sampled PC centroid to track instruction working set and coarse-grain phase changes.

Phase Aware Profiling. Nagpurkar et al [16] proposed a flexible hardware-software scheme for efficient remote profiling on networked embedded device. It relies on the extraction of meta information from executing programs in the form of phases, and then use this information to guide intelligent online sampling and to manage the communication of those samples. They used BBV based hardware phase tracker which was proposed in [2] and enhanced in [5].

3 Dynamic Code Region Based Program Phase Tracking

3.1 Tracking Dynamic Code Region as a Phase

In this paper, we propose phase tracking hardware that only tracks functions and loops in the program. The hardware consists of a stack and a phase history table. The idea of using a hardware stack is based on the observation that any path from the root node to a node representing a dynamic code region in *Extended Calling Context Tree (ECCT)* can be represented as a stack. To illustrate this observation, we use an example program and its corresponding ECCT in Figure 1. In this example, we assume that each of loop0, loop1 and loop3 executes for a long period of time, and represents dynamic code regions that are potential targets for optimizations. The sequence of function calls which leads to loop1 is main() → func1() → loop1. Thus, if we maintain a runtime stack of the called functions and executed loops while loop1 is executing, we would have main(), func1() and loop1 on it. Similarly, as shown in Figure 1, the content of the stack for code region 2 would be main() and loop0, while for code region 3 it would be main(), func3() and loop3. They uniquely identify the calling context of a code region, and thus could be used as a signature that identifies the code region. For example, the code region loop3 could be identified by the signature main() → func3() on the stack. Code regions in Figure 1 can be formed during runtime. This is why it is called *Dynamic Code Region (DCR)*. Stable DCR is a sub tree which has a stable calling context in ECCT during a monitoring interval, such as one million instructions.

The phase signature table extracts information from the stack and stores the stack signatures. It also assigns a phase ID for each signature. The details of the fields in the table and its function are presented in section 4.

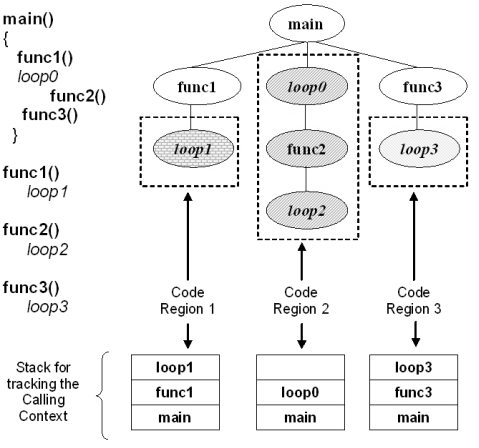
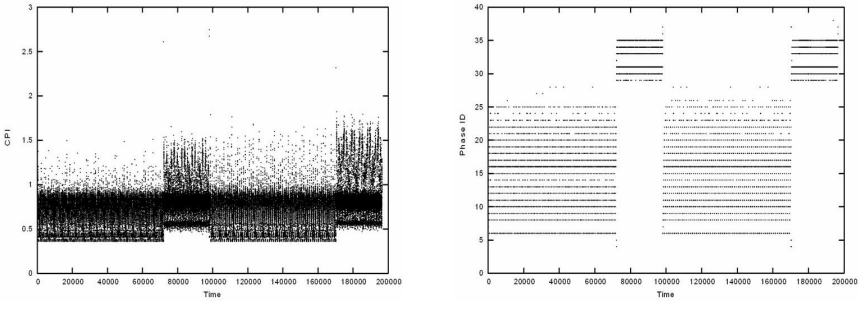


Fig. 1. An example code and its corresponding ECCT representation. Three dynamic code regions are identified in the program and are marked by different shades in the tree.

3.2 Correlation Between Dynamic Code Regions and Program Performance Behaviors

In Figure 2(a), CPI calculated for every 1-million-instruction interval for bzip2 is plotted. We then used the notion of DCR also for every one million instructions and assigned a distinct phase ID for each DCR. Such ID's are then plotted against the time shown in Figure 2(b). Comparing the CPI graph in (a) with the phase ID graph in (b), it can be seen that the CPI variation in the program has a strong correlation with changes in DCR's. This shows that DCR's in a program could reflect program performance behavior and tracks the boundaries of behavior changes. Although BBV also shows the similar correlation, DCR



(a) CPI change over the time (b) Phase Change over the time

Fig. 2. Visualizing phase change in bzip2. (a) Change of average CPI. (b) Tracking Phase changes. The Y-axis is phase ID.

gives code regions that aligned with procedures and loops, which exhibits a higher accuracy in phase tracking and also make it easier for optimization.

There are several horizontal lines in Figure 2(b). It shows that a small number of DCR's are being repeatedly executed during the period. Most DCR's seen in this program are loops. More specifically, phase ID 6 is a loop in `loadAndRLLSource`, phase ID 10 is a loop in `SortIt`, phase ID 17 is a loop in `generateMTFvalues`, and phase ID 31 is a loop in `getAndMoveTofrontDecode`.

3.3 Phase Detection for Dynamic Optimization Systems

Dynamic optimization systems prefer longer phases. If the phase detector is overly sensitive, it may trigger profiling and optimization operations too often to cause performance degradation. Phase prediction accuracy is essential to avoid bringing unrelated traces into the code cache. The code cache management system would also require information about the phase boundaries to precisely identity the code region for the phase. The information about the code structure can be used by the profiler and the optimizer to identify the code region for operation. Finally, it is generally beneficial to have a small number of phases as long as we can capture most important program behavior; this is because a small number of phases allow the phase detector to identify longer phases and to predict the next phase more accurately. It should be noted that dynamic optimization systems can trade a little variability within each phase for longer phase length and higher predictability, as these factors determines the overheads of the system.

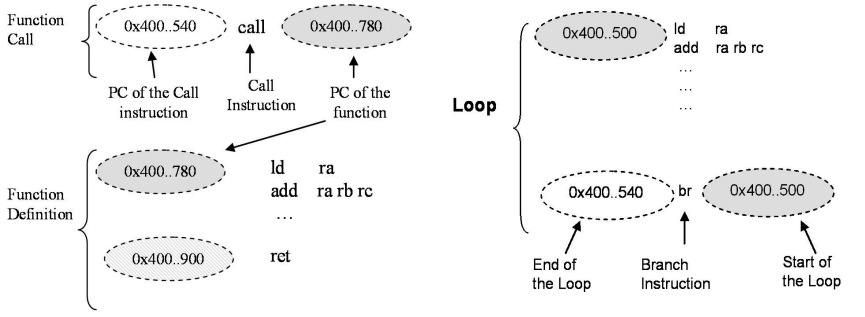
4 DCR-Based Phase Tracking and Prediction Hardware

We have discussed why DCR can be used to track program execution phases. In this section, we propose a relatively simple hardware structure to track DCR during program execution.

4.1 Identifying Function Calls and Loops in the Hardware

Function calls and their returns are identified by call and ret instructions in the binary. Most modern architectures have included call/ret instructions defined. On detecting a call instruction (see Figure 3(a)), the PC of the call instruction and the target address of the called function are pushed onto the hardware stack. On detecting a return instruction, they are popped out of the stack. A special case to be considered when detecting function calls and return is recursion. In section 4.3 we describe a technique to deal with recursions.

Loops can be detected using backward branches. A branch which jumps to an address that is lower than the PC of the branch instruction is a backward branch. The target of the branch is the start of the loop and the PC of the branch instruction is the end of the loop. This is illustrated in Figure 3(b). These two addresses represent the boundaries of a loop. Code re-positioning transformations



(a) function call (b) loop

Fig. 3. Assembly code of a function call (a) and loop (b). The target address of the branch instruction is the start of the loop and the PC address of the branch instruction is the end of the loop.

can introduce backward branches that are not loop branches. Such branches may temporarily be put on the stack and then get removed quickly. On identifying a loop, the two addresses marking the loop boundaries are pushed onto the stack. To detect a loop, we only need to detect the first iteration of the loop. In order to prevent pushing these two addresses onto the stack multiple times in the subsequent iterations, the following check is performed. On detecting a backward branch, the top of the stack is checked to see if it is a loop. If so, the addresses stored at the top of the stack are compared to that of the detected loop. If the addresses match, we have detected an iteration of a loop which is already on the stack. A loop exit occurs when the program branches out to an address outside the loop boundaries. On a loop exit, the loop node is popped out of the stack.

4.2 Hardware Description

The schematic diagram of the hardware is shown in Figure 4. The central part of the phase detector is a hardware stack and the signature table. The configurable parameters in the hardware are the number of entries in the stack and the number of entries in the phase signature table.

Each entry in the hardware stack consists of four fields. The first two fields hold different information for functions and loops. In the case of a function, the first and second fields are used to store the PC address of the call instruction and the PC of the called function respectively. The PC address of the call instruction is used in handling recursions. In the case of a loop, the first two fields are used to store the start and end address of the loop. The third field is a one bit value, called the stable stack bit. This bit is used to track the signature of the dynamic code region. At the start of every interval, the stable stack bit for all entries in the stack which holds a function or a loop is set to '1'. The stable stack bit remains zero for any entry that is pushed into or popped out of the stack during the interval. At the end of the interval, those entries in the bottom of the stack whose stable stack bit is still '1' are entries which were not popped

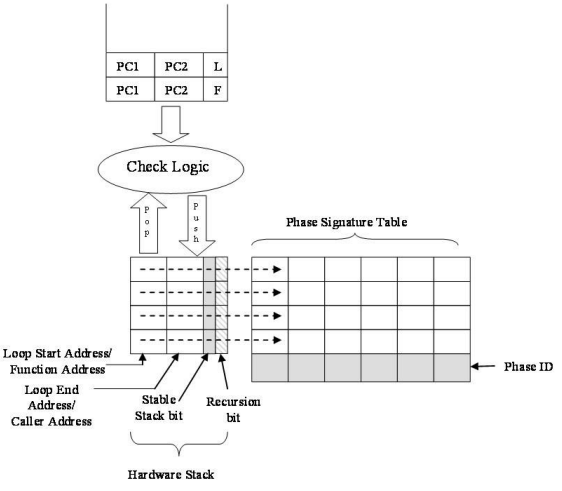


Fig. 4. Schematic diagram of the hardware phase detector

during the current interval. These set of entries in the bottom of the stack form the signature to the region in the code to which the execution was restricted to, in the current interval. At the end of the interval, this signature is compared against all signatures stored in the phase signature table. The phase signature table holds the stack signature seen in the past and its associated phase ID. On a match, the phase ID corresponding to that entry in the signature table is returned as the current phase ID. If a match is not detected, a new entry is created in the signature table with the current signature and a new phase ID is assigned to it. If there are no free entries available in the signature table, the least recently used entry is evicted from the signature table to create space for the new entry. The fourth field is a one bit value called the recursion bit and is used when handling recursions. The use of this bit is explained in section 4.3.

4.3 Handling Special Cases

Recursions. In our phase detection technique, all functions that form a cycle in the call graph (i.e., they are recursive calls) are considered as members of the same dynamic code region.

In our hardware, all recursions are detected by checking the content of the stack. A recursion is detected when the address of the function being called is already present in an entry on the stack. This check assumes that an associative lookup of the stack is done during every push operation. Since the number of entries on the stack is small, the associative lookup hardware would be feasible.

To avoid stack overflow during a recursion, no push operation is performed after a recursion is detected. The recursion bit is set to '1' for the entry corresponding to the function marking the boundary of the recursion. Since we no longer push any entry onto the stack, we cannot pop any entry from the stack on

detecting a return instruction, until it is out of the recursion cycle. This can be detected when a return instruction jumps to a function outside of the recursion cycle. All entries in the stack that are functions, which lies below the entry whose recursion bit is set, are outside the recursion cycle. After a recursion is detected, the return address of all subsequent return instructions are checked against these entries. On a match, all entries above the matched entry are flushed, and normal stack operation is resumed.

Hardware Stack Overflow. Recursion is not the only case in which a stack overflow could occur. If the stack signature of a dynamic code region has more elements than the stack can hold, the stack would overflow. We did not encounter any stack overflow for a 32-entry stack. But if it did occur, it is handled very similar to a recursive call described earlier. On a stack overflow, no further push operation is performed. The address to which the control gets transferred during a return instruction is checked for a match to an address in the stack. If it matches, all entries above this instruction are removed from the stack and normal stack operation is resumed.

5 Evaluation

5.1 Evaluation Methodology

Pin and pfmon [9,11] were used in our experiments on evaluating the effectiveness of the phase detector. Pin is a dynamic instrumentation framework developed at Intel for Itanium processors [11]. A Pin instrumentation routines was developed to detect function calls and returns, backward branches for loops and to maintain a runtime stack.

pfmon is a tool which reads performance counters of the Itanium processor [11]. We use CPI as the overall performance metric to analyze the variability within detected phases. We modified pfmon to get CPI for every one million instructions. To minimize random noise in our measurements,the data collection was repeated 3 times and the average of the 3 runs was used for the analysis. These CPI values were then matched with the phase information obtained from the Pin Tool, to get the variability information.

All the data reported in this paper were collected from a 900 Mhz Itanium-2 processor with 1.5 M L3 cache running Redhat Linux Operating System version 2.4.18-e37.

5.2 Metrics

The metrics used in our study are the number of distinct phases, average phase length, Coefficient of Variance (CoV) of CPI, and ratio of next phase prediction. The number of distinct phases corresponds to the number of dynamic code regions detected in the program. Average phase length of a benchmark program gives the average number of contiguous intervals classified into a phase. It is calculated by taking the sum of the number of contiguous intervals classified into a

phase divided by the total number of phases detected in the program. Coefficient of Variation quantifies the variability of program performance behavior and is given by

$$CoV = \frac{\sigma}{\mu} \tag{1}$$

CoV provides a relative measure of the dispersion of data when compared to the mean. We present a weighted average of CoV on different phases detected in each program. Weighted average of the CoV gives more weight to the CoV of phases that have more intervals (i.e., longer execution times) in it and hence, better represent the CoV observed in the program.

The ratio of next phase prediction is the number of intervals whose phase ID was correctly predicted by the hardware divided by the total number of intervals in the program.

5.3 Benchmarks

Ten benchmarks from the SPEC CPU2000 benchmark suite (8 integer and 2 floating point benchmarks) were evaluated. These benchmarks were selected for this study because they are known to have interesting phase behavior and are challenging for phase classification [2]. Reference input sets were used for all benchmarks. Three integer benchmarks, namely gzip, bzip and vpr, were evaluated with 2 different input sets. A total of 13 benchmarks and input sets combinations were evaluated. All benchmarks were compiled using gcc (version 3.4) at O3 optimization level.

6 Experimental Results

In this section, we present the results of our phase classification and prediction hardware. There are two configurable parameters in our hardware. They are the size of the stack and the size of the phase history table. We evaluated four different configurations for the hardware. The size of the stack and the size of the phase history table were set to 16/16, 32/64, 64/64 and infinity/infinite respectively. We found no significant differences between them in the metrics described above. Due to lack of space, we are unable to show the results here. Interested readers can refer to [28] for the analysis.

In 6.1, we compare the results of our hardware scheme to the basic block vector (BBV) scheme described in [2,5]. A Pin tool was created to get the BBV. These vectors were analyzed off-line using our implementation of the phase detection mechanism described in [2,5] to generate the phase ID's for each interval. The metrics which we use for comparison include: the total number of phases detected, average phase length, the next phase prediction accuracy and the CoV of the CPI within each phase.

6.1 Comparison with BBV Technique

In this section, we compare the performance of our phase detection hardware with the phase detection hardware based on BBV [2,5]. The BBV-based phase

detection hardware is described in [2]. There are 2 tables in the hardware structure, namely the accumulator table which stores the basic block vector and the signature table which stores the basic block vectors seen in the past. These structures are similar in function to our hardware stack and signature table, respectively. To make a fair comparison, we compare our 32/64 configuration against a BBV-based hardware which has 32 entries in the accumulator table and 64 entries in the signature table.

We compare our phase detector and BBV based detector using four parameters namely, number of phases detected, phase length, predictability of phases, and stability of the performance within each phase. We compare our results with those of the BBV technique for two threshold values namely 10% and 40% of an one-million-instruction interval. The original BBV paper [2] sets the threshold value to be 10% of the interval. It should be noted that the phases detected in the BBV-based technique may not be aligned with the code structures. Aligned phases are desirable for Dynamic Binary Re-Optimization System. We still compare against the BBV method because it is a well accepted method for detecting phases [26].

Number of Phases and Phase Length. Table 1 compares the number of phases detected for the BBV technique and our phase detection technique. For the BBV technique, there are 2 columns for each benchmark that correspond to a threshold value of 10% and 40% of one million instructions, respectively. In the case of BBV technique, as we increase the threshold value, small differences between the basic block vectors will not cause phase change. Hence, less number of phases is detected as we go from 10% to 40% threshold value. Recall that, in

Table 1. Comparison of the number of phases and length of phases detected between BBV- and DCR-based phase detection schemes. A 32-entry accumulator table/hardware stack and a 64-entry phase signature table were used.

	Number of phases			Length of phases		
Benchmarks	BBV-10%	BBV-40%	DCR-32/64	BBV-10%	BBV-40%	DCR-32/64
ammp	13424	122	53	58.83	6472.27	15027.77
bzip2_1	35154	1796	99	5.59	111.51	1984.60
bzip2_2	37847	1469	87	4.24	108.30	1845.51
crafty	20111	20	27	14.57	15073.26	10314.54
eon	38	7	22	6128.94	31520.43	10029.18
gcc	2650	599	337	19.70	86.84	158.15
gzip_1	8328	182	48	13.90	629.34	2450.40
gzip_2	4681	77	42	11.11	678.23	1273.73
mcf	5507	88	55	19.52	1219.19	1950.67
mesa	945	15	37	517.82	32899.13	13402.89
perl	8036	201	28	13.76	497.59	3579.22
vpr_1	3136	105	91	47.17	1398.50	1618.96
vpr_2	51	27	27	3715.51	7018.19	7018.19
median	5507	105	48	19.52	1219.19	2450.40

a Dynamic Binary Optimization system, on detecting a new phase, the profiler will start profiling the code, which will cause significant overhead. Hence, for such systems less number of phases with longer per phase length is desirable. We can see that in the original BBV technique with 10% threshold, the number of phases detected is 100 times more than those detected in the DCR based technique. In the BBV technique as we go from 10% to 40% threshold value, the number of phases detected becomes smaller, which is expected. But even at 40%, the number of phases detected in BBV technique is 2x more than those detected by our technique. Table 1 also shows the average phase lengths for the BBV technique and our phase detection technique. The median phase length of our technique is 100 times more than those found in BBV technique with 10% threshold value, and two times more than those found in BBV technique with 40% threshold value. Although in the case of `eon` and `mesa` the phase length for BBV with 40% threshold value is 3 times that of DCR technique, these programs are known to have trivial phase behavior. The larger difference is due to difference in the number of phases detected in our case.

Performance Variance Within Same Phase. Figure 5 compares the performance of the 256-entry Markov Predictor using BBV technique and our phase detection technique. Except eon and vpr_1, the Markov predictor predicts the next phase better using our phase detector. On average using our phase detection technique, the Markov predictor predicts the correct next phase ID 84.9% of the time. Using BBV-based technique, the average correct prediction ratios are 42.9% and 73.3% for 10% and 40% respectively.

Phase Prediction Accuracy. Figure 6 compares the weighted average of the CoV of CPI for phases detected by the BBV technique and by our phase detection technique. The last four bars give the average CoV. From the figure we can see that BBV-10% has the least average CoV value. This is because the number of

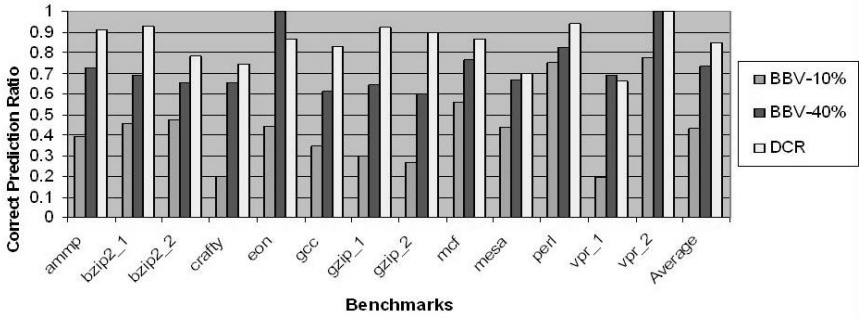


Fig. 5. Comparison between BBV- and DCR-based phase detection A 32-entry accumulator table/hardware stack and a 64-entry phase signature table were used. The first 2 columns for each benchmark are for BBV method using threshold values of 10% and 40% of one million instructions respectively.

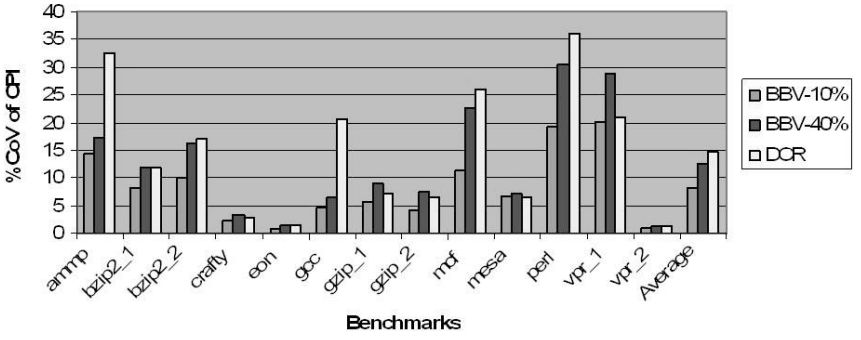


Fig. 6. Comparison of the weighted average of the CoV of CPI between BBV- and DCR-based phase detection schemes. A 32-entry accumulator table/hardware stack and a 64-entry phase signature tables were used.

phases detected by BBV-10% is much higher than the number of phases detected by BBV-40% or our technique. In the case of BBV-10%, the variability of CPI gets divided into many phases, thus reducing the variability observed per phase. On average the %CoV of our phase detection hardware is 14.7% while it is 12.57% for the BBV-40%. Although the average variability of our technique is greater than BBV-40%, the numbers are comparable. In fact for bzip2_1, crafty, gzip_1, gzip_2, mesa, vpr_1 and vpr_2, the CoV of the DCR based technique is less than or equal to the CoV observed for the BBV-40% . For ammp, gcc, mcf and perl the performance variation is higher in the dynamic code regions detected. The higher performance variation within each dynamic code region may be due to change in control flow as in the case of gcc or change in data access patterns as in the case of mcf.

From the above discussions we can conclude that, our hardware detects less number of phases, has a longer average phase length, has higher phase predictability and is aligned with the code structure, all of which are desirable characteristics of a phase detector for a Dynamic Binary Re-optimization system. The CoV of the phases detected in our technique is higher but comparable to that observed in the BBV technique with 40% threshold value. The phase difference is detected using an absolute comparison of phase signatures, which makes the hardware simpler and the decision easier to make. The 32/64 hardware configuration performs similar to an infinite sized hardware, which makes it cost effective and easier to design.

7 Conclusion and Future Work

We have evaluated the effectiveness of our DCR-based phase tracking hardware on a set of SPEC benchmark programs with known phase behaviors. We have shown that our hardware exhibits the desirable characteristics of a phase detector

for dynamic optimization systems. The hardware is simple and cost effective. The phase sequence detected by our hardware could be accurately predicted using simple prediction techniques. We are currently implementing our phase detection technique within a dynamic optimization framework. We plan to evaluate the effectiveness of our technique for dynamic profile guided optimizations such as instruction cache prefetching and for code cache management.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments. This work was supported in part by the National Science Foundation under Grant No. EIA-0220021, Intel and the Minnesota Supercomputing Institute.

References

1. A. Dhodapkar and J.E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In 29th Annual International Symposium on Computer Architecture, May 2002.
2. T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In 30th Annual International Symposium on Computer Architecture, June 2003.
3. E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In International Conference on Parallel Architectures and Compilation Techniques, October, 2003
4. X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In International Conference on Architectural Support for Programming Languages and Operating Systems, 2004
5. J. Lau, S. Schoenmackers, and B. Calder. Transition Phase Classification and Prediction, In the 11th International Symposium on High Performance Computer Architecture, February, 2005.
6. M. Sun, J.E. Daly, H. Wang and J.P. Shen. Entropy-based Characterization of Program Phase Behaviors. In the 7th Workshop on Computer Architecture Evaluation using Commercial Workloads, February 2004.
7. J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, and P.-C. Yew. The Performance of Data Cache Prefetching in a Dynamic Optimization System. In the 36th International Symposium on Microarchitecture, December 2003.
8. M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In 30th Annual International Symposium on Computer Architecture, June 2003.
9. PIN - A Dynamic Binary Instrumentation Tool. http://rogue.colorado.edu/Pin.
10. J. Lau, S. Schoenmackers, and B. Calder. Structures for Phase Classification. In IEEE International Symposium on Performance Analysis of Systems and Software, March 2004.
11. http://www.hpl.hp.com/research/linux/perfmon.
12. V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2000.

13. K. Hazelwood and M.D. Smith. Generational cache management of code traces in dynamic optimization systems. In 36th International Symposium on Microarchitecture, December 2003.
14. K. Hazelwood and James E. Smith. Exploring Code Cache Eviction Granularities in Dynamic Optimization Systems. In second Annual IEEE/ACM International Symposium on Code Generation and Optimization, March 2004.
15. M. Arnold and D. Grove. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In third Annual IEEE/ACM International Symposium on Code Generation and Optimization, March 2005.
16. P. Nagpurkar, C. Krintz and T. Sherwood. Phase-Aware Remote Profiling. In the third Annual IEEE/ACM International Symposium on Code Generation and Optimization, March 2005.
17. D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In First Annual International Symposium on Code Generation and Optimization, March 2003.
18. W.-K. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization. In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization, 2000.
19. S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufman, 1997.
20. H. Chen, J. Lu, W.-C Hsu, P.-C Yew. Continuous Adaptive Object-Code Reoptimization Framework, In 9th Asia-Pacific Computer Systems Architecture Conference, 2004
21. M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeno JVM, in Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, October 2000.
22. M.J. Hind, V.T. Rajan, and P.F. Sweeney. Phase shift detection: a problem classification, in IBM Research Report RC-22887, pp. 45-57.
23. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI), June 1997.
24. G. Magklis, M. L. Scott, G. Semeraro, D. A. Albonesi, and S. Dropsho, Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor. In Proceedings of the International Symposium on Computer Architecture, June 2003.
25. C.-H Hsu and U. Kermer. The design, implementation and evaluation of a compiler algorithm for CPU energy reduction. In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2003.
26. A. S. Dhodapkar and J. E. Smith, Comparing program phase detection techniques. In the 36th International Symposium on Microarchitecture, December 2003.
27. W. Liu and M. Huang, EXPERT: expedited simulation exploiting program behavior repetition. In Proceedings of the 18th annual international conference on Supercomputing, June 2004.
28. J. Kim, S.V. Kodakara, W.-C. Hsu, D.J. Lilja and P.-C Yew, Dynamic Code Region-based Program Phase Classification and Transition Prediction, in Uinversity of Minnesota, Computer Science & Engineering Technical Report 05-021, May 2005.

Induction Variable Analysis with Delayed Abstractions

Sebastian Pop[1], Albert Cohen[2], and Georges-André Silber[1]

[1] CRI, Mines Paris, Fontainebleau, France
[2] ALCHEMY group, INRIA Futurs, Orsay, France

Abstract. This paper presents the design of an induction variable analyzer suitable for the analysis of typed, low-level, three address representations in SSA form. At the heart of our analyzer is a new algorithm recognizing scalar evolutions. We define a representation called trees of recurrences that is able to capture different levels of abstractions: from the finer level that is a subset of the SSA representation restricted to arithmetic operations on scalar variables, to the coarser levels such as the evolution envelopes that abstract sets of possible evolutions in loops. Unlike previous work, our algorithm tracks induction variables without prior classification of a few evolution patterns: different levels of abstraction can be obtained on demand. The low complexity of the algorithm fits the constraints of a production compiler as illustrated by the evaluation of our implementation on standard benchmark programs.

1 Introduction and Motivation

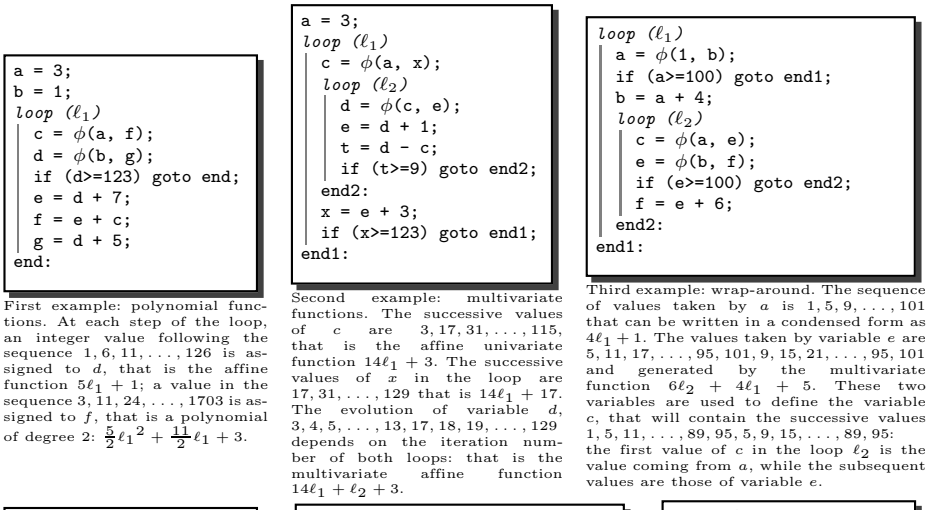
Supercomputing research has produced a wealth of techniques to optimize a program and tune code generation for a target architecture, both for uniprocessor and multiprocessor performance [35,2]. But the context is different when dealing with common retargetable compilers for general-purpose and/or embedded architectures: the automatic exploitation of parallelism (fine-grain or thread-level) and the tuning for dynamic hardware components become far more challenging. Modern compilers implement some of the sophisticated optimizations introduced for supercomputing applications [2], provide performance models and transformations to improve fine-grain parallelism and exploit the memory hierarchy. Most of these optimizations are loop-oriented and assume a high-level code representation with rich control and data structures: `do` loops with regular control, constant bounds and strides, typed arrays with linear subscripts. Good optimizations require manual efforts in the syntactic presentation of loops and array subscripts (avoiding, e.g., `while` loops, pointers, exceptions, or goto statements). Programs written for embedded systems often make use of such low-level constructs, and this programming style is not suited to traditional source-to-source loop nest optimizers. Because any rewrite of the existing code involves an important investment, we see the need of a compiler that could optimize low-level constructs. Several works demonstrated the interest of enriched low-level representations [15,23]: they build on the normalization of three address code, adding

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 218–232, 2005.

data types, *Static Single-Assignment* form (SSA) [6,19] to ease data-flow analysis and scalar optimizations. Starting from version 4.0, GCC uses such a representation: GIMPLE [21,17], a three-address code derived from SIMPLE [12]. In a three-address representation, subscript expressions, loop bounds and strides are spread across several instructions and basic blocks. The most popular techniques to retrieve scalar evolutions [34,8,27] are not well suited to work on loosely structured loops because they rely on classification schemes into a set of predefined forms based on pattern-matching rules. Such rules are sometimes sufficient at the source level, but too restrictive to cover the wide variability of inductive schemes on a low-level representation. To address the challenges of induction variable recognition on a low-level representation, we designed a general and flexible algorithm to build closed form expressions for scalar evolutions. This algorithm can retrieve array subscripts, loop bounds and strides lost in the lowering to three-address code, as well as other properties that do not appear in the source code. We demonstrate that induction-variable recognition and dependence analysis can be effectively implemented at such a low level. We also show that our method is more flexible and robust than comparable solutions on high-level code [8,32,31], since our method captures affine and polynomial closed forms without restrictions on the complexity of the flow of control, the recursive scalar definitions, and the intricateness of ϕ nodes. Finally, speed, robustness and language-independence are natural benefits of a low-level SSA representation.

Intermediate Representations. We recall some SSA terminology, see [6,19] for details: the *SSA graph* is the graph of def-use chains; ϕ nodes occur at merge points and restore the flow of values from the renamed variables; ϕ nodes are split into the *loop-ϕ* and *condition-ϕ* nodes. In this paper, we use a typed three-address code in SSA form. Control-flow primitives are a conditional expression `if`, a goto expression, and a loop annotation discovered from the control-flow graph: *loop (ℓ_k)* stands for loop number k, and ℓ_k denotes its associated implicit counter. The number of iterations is computed from the evolutions of scalars in the loop exit conditions, providing informations lost in the translation to a low-level, or not exposed at source level, as in `while` or `goto` loops.

Introductory Examples. To illustrate the main issues and concepts, we consider the examples in Figure 1. A closed-form for `f` in the first example is a second-degree polynomial. In the second example, `d` has a *multivariate* evolution: it depends on several loop counters. To compute the evolution of `c`, `x` and `d` in the second example, one must know the *trip count* of the inner loop, here 10 iterations. Yet, to statically evaluate the trip count of ℓ_2 one must already understand the evolutions of `c` and `d`. In the third example, `c` is a typical case of *wrap-around* variable [34]. In the fourth example `a` and `b` have linear closed forms. Unlike our algorithm, previous works could not compute this closed form due to the intricateness of the SSA graph. The fifth example illustrates a data dependence problem: when language standards define modulo arithmetics for a type, the compiler has to respect effects of overflows, and otherwise, as in the sixth example, the compiler can deduce constraints from undefined behavior.



First example: polynomial functions. At each step of the loop, an integer value following the sequence $1, 6, 11, \ldots, 126$ is assigned to d, that is the affine function $5\ell_1 + 1$; a value in the sequence $3, 11, 24, \ldots, 1703$ is assigned to f, that is a polynomial of degree 2: $\frac{5}{2}{\ell_1}^2 + \frac{11}{2}\ell_1 + 3$.

Second example: multivariate functions. The successive values of c are $3, 17, 31, \ldots, 115$, that is the affine univariate function $14\ell_1 + 3$. The successive values of x in the loop are $17, 31, \ldots, 129$ that is $14\ell_1 + 17$. The evolution of variable d, $3, 4, 5, \ldots, 13, 17, 18, 19, \ldots, 129$ depends on the iteration number of both loops: that is the multivariate affine function $14\ell_1 + \ell_2 + 3$.

Third example: wrap-around. The sequence of values taken by a is $1, 5, 9, \ldots, 101$ that can be written in a condensed form as $4\ell_1 + 1$. The values taken by variable e are $5, 11, 17, \ldots, 95, 101, 9, 15, 21, \ldots, 95, 101$ and generated by the multivariate function $6\ell_2 + 4\ell_1 + 5$. These two variables are used to define the variable c, that will contain the successive values $1, 5, 11, \ldots, 89, 95, 5, 9, 15, \ldots, 89, 95$: the first value of c in the loop ℓ_2 is the value coming from a, while the subsequent values are those of variable e.

Fourth example : periodic functions. Both a and b have affine evolutions: $0, 1, 2, \ldots, 100$, because they both have the same initial value. However, if their initial value is different, their evolution can only be described by a periodic affine function.

Fifth example: effects of types on the evolution of scalar variables. The C programming language defines modulo arithmetics for unsigned typed variables. In this example, the successive values of variable a are periodic: $0, 1, 2, \ldots, 255, 0, 1, \ldots$, or in a condensed notation $\ell_1 \ mod \ 256$.

Sixth example: inferring properties from undefined behavior. Signed types overflow are not defined in C. The behavior is only defined for the values of a in $0, 1, 2, \ldots, 126$, consequently d is only defined for $1, 2, 3, \ldots, 127$, and the loop is defined only for the first 127 iterations.

Fig. 1. Examples

Overview of the Paper. In the following, we expose a set of techniques to extract and to represent evolutions of scalar variables in the presence of complex control flow and intricate inductive definitions. We focus on designing low-complexity algorithms that do not sacrifice on the effectiveness of retrieving precise scalar evolutions, using a typed, low-level, SSA-form representation of the program. Section 2 introduces the algebraic structure that we use to capture a wide spectrum of scalar evolution functions. Section 3 presents the analysis algorithm to extract closed form expressions for scalar evolutions. Section 4 compares our method to other existing approaches. Finally, section 5 concludes and sketches future work. For space constraints, we have shortened this presentation. A longer version of the paper is available as a technical report [26].

2 Trees of Recurrences

In this section, we introduce the notion of *Tree of Recurrences* (TREC), a closed-form that captures the evolution of induction variables as a function of iteration

indices and allows an efficient computation of values at given iteration points. This formalism extends the expressive power of *Multivariate Chains of Recurrences* (MCR) [3,14,36,32] by symbolic references. MCR are obtained by an abstraction operation that instantiate all the varying symbols: some evolutions are mapped to a "don't know" symbol $\top$. Arithmetic operations on MCR are defined as rewriting rules [32]. Let $F(\ell_1, \ell_2, \ldots, \ell_m)$, or $F(\boldsymbol{\ell})$, represent the evolution of a variable inside a loop of depth m as a function of $\ell_1, \ell_2, \ldots, \ell_m$. F can be written as a closed form Θ, called TREC, that can be statically processed by further analyzes and efficiently evaluated at compile-time. The syntax of a TREC is derived from MCR and inductively defined as: $\Theta = \{\Theta_a, +, \Theta_b\}_k$ or $\Theta = c$, where Θ_a and Θ_b are trees of recurrences and c is a constant or a variable name, and subscript k indexes the dimension. As a form of syntactic sugar, $\{\Theta_a, +, \{\Theta_b, +, \Theta_c\}_k\}_k = \{\Theta_a, +, \Theta_b, +, \Theta_c\}_k$.

Evaluation of TREC. The value $\Theta(\ell_1, \ell_2, \ldots, \ell_m)$ of a TREC Θ is defined as follows: if Θ is a constant c then $\Theta(\boldsymbol{\ell}) = c$, else Θ is $\{\Theta_a, +, \Theta_b\}_k$ and

$$\Theta(\boldsymbol{\ell}) = \Theta_a(\boldsymbol{\ell}) + \sum_{l=0}^{\ell_k - 1} \Theta_b(\ell_1, \ldots, \ell_{k-1}, l, \ell_{k+1}, \ldots, \ell_m) \ .$$

The evaluation of $\{\Theta_a, +, \Theta_b\}_k$ for a given $\boldsymbol{\ell}$ matches the inductive updates across ℓ_k iterations of loop k: Θ_a is the initial value, and Θ_b the increment in loop k. This is an exponential algorithm to evaluate a TREC, but [3] gives a linear time and space algorithm based on Newton interpolation series. Given a univariate MCR with $c_0, c_1, \ldots, c_n$, constant parameters (either scalar constants, or symbolic names defined outside loop k):

$$\{c_0, +, c_1, +, c_2, +, \ldots, +, c_n\}_k(\boldsymbol{\ell}) = \sum_{p=0}^{n} c_p \binom{\ell_k}{p} \ . \tag{1}$$

This result comes from the following observation: a sum of multiples of binomial coefficients — called *Newton series* — can represent any polynomial. The closed form for `f` in the first example of Figure 1 is the second order polynomial $F(\ell_1) = \frac{5}{2}{\ell_1}^2 + \frac{11}{2}\ell_1 + 3 = 3\binom{\ell_1}{0} + 8\binom{\ell_1}{1} + 5\binom{\ell_1}{2}$, and is written $\{3, +, 8, +, 5\}_1$. The coefficients of a TREC derive from a finite differentiation table: for example, the coefficients for the TREC associated with $\frac{5}{2}{\ell_1}^2 + \frac{11}{2}\ell_1 + 3$ can be computed either by differencing the successive values [11]:

ℓ_1	0	1	2	3	4
c_0	**3**	11	24	42	65
c_1	**8**	13	18	23	
c_2	**5**	5	5		
c_3	**0**	0			

or, by directly extracting the coefficients from the code [32]. We present our algorithm for extracting TREC from a SSA representation in Section 3. We illustrate the fast evaluation of a TREC from the second introductory example Figure 1, where the evolution of `d` is the affine equation $F(\ell_1, \ell_2) = 14\ell_1 + \ell_2 + 3$. A TREC for `d` is $\Theta(\ell_1, \ell_2) = \{\{3, +, 14\}_1, +, 1\}_2$, that can be evaluated for $\ell_1 = 10$ and $\ell_2 = 15$ as follows:

$$\Theta(10,15) = \{\{3,+,14\}_1,+,1\}_2(10,15) = 3 + 14 \cdot \binom{10}{1} + \binom{15}{1} = 158 \ .$$

Instantiation of TREC and Abstract Envelopes. In order be able to use the efficient evaluation scheme presented above, symbolic coefficients of a TREC have to be analyzed: the role of the instantiation pass is to limit the expressive power of TREC to MCR. Difficult TREC constructs such as exponential self referring evolutions (as the Fibonacci sequence that defines the simplest case of the class of mixers: $fib \rightarrow \{0,+,1,+,fib\}_k$) are either translated to some appropriate representation, or discarded. Optimizers such as symbolic propagation could handle such difficult constructs, however they lead to problems that are difficult to solve (e.g. determining the number of iterations of a loop whose exit edge is guarded by a Fibonacci sequence). Because a large class of optimizers and analyzers are expecting simpler cases, TREC information is filtered using an instantiation pass. Several abstract views can be defined by different instantiation passes, such as mapping every non polynomial scalar evolution to $\top$, or even more practically, mapping non affine functions to $\top$. In appropriate cases, it is natural to map uncertain values to an abstract value: we have experimented instantiations of TREC with intervals, in which case we obtain a set of possible evolutions that we call an envelope. Allowing the coefficients of TREC to contain abstract scalar values is a more natural extension than the use of maximum and minimum functions over MCR as proposed by van Engelen in [31] because it is then possible to define other kinds of envelopes using classic scalar abstract domains, such as polyhedra, octagons [18], or congruences [10].

Peeled Trees of Recurrences. A frequent occurring pattern consists in variables that are initialized to a value during the first iteration of a loop, and then is replaced by the values of an induction variable for the rest of iterations. We have chosen to represent these variables by explicitly listing the first value that they contain, and then the evolution function that they follow. The *peeled* TREC are described by the syntax $(a,b)_k$ whose semantics is given by:

$$(a,b)_k(x) = \begin{cases} \text{a} & \text{if } x = 0, \\ \text{b (x - 1)} & \text{for } x \geq 1, \end{cases}$$

where a is a TREC with no evolution in loop k, b is a TREC that can have an evolution in loop k, and x is indexing the iterations in loop k. Most closed forms for wrap-around variables [34] are peeled TREC. Indeed, back to the third introductory example (see Figure 1), the closed form for c can be represented by a peeled multivariate affine TREC: $(\{1,+,4\}_1, \{\{5,+,4\}_1,+,6\}_2)_2$. A peeled TREC describes the first values of a closed form chain of recurrence. In some cases it is interesting to replace it by a simpler MCR, and vice versa, to peel some iterations out of a MCR. For example, the peeled TREC $(0,\{1,+,1\}_1)_1$ describes the same function as $\{0,+,1\}_1$. This last form is a unique representative of a class of TREC that can be generated by peeling one or more elements from the beginning. Simplifying a peeled TREC amounts to the unification of its first element with the function

ℓ_1	0	1	2	3	4
c_0	3	11	24	42	65
c_1	8	13	18	23	
c_2	5	5	5		
c_3	0	0			

ℓ_1	0	1	2	3	4	5
c_0	**0**	3	11	24	42	65
c_1	**3**	8	13	18	23	
c_2	**5**	5	5	5		
c_3	**0**	0	0			

Fig. 2. Adding a new column to the differentiation table of the chain of recurrence $\{3, +, 8, +, 5\}_1$ leads to the chain of recurrence $\{0, +, 3, +, 5\}_1$

represented in the right-hand side of the peeled TREC. A simple unification algorithm tries to add a new column to the differentiation table without changing the last element in that column. Since this first column contains the coefficients of the TREC, the transformation is possible if it does not modify the last coefficient of the column, as illustrated in Figure 2. This technique allows to unify 29 wrap around loop-ϕ in the SPEC CPU2000, 337 on the GCC code itself, and 5 on the JavaGrande. Finally, we formalize the notion of peeled TREC equivalence class: given integers $v, a_1, \ldots, a_n$, a TREC $c = \{a_1, +, \ldots, +, a_n\}_1$, a peeled TREC $p = (v, c)_1$, and a TREC $r = \{b_1, +, \ldots, +, b_{n-1}, +, a_n\}_1$, with the integer coefficients $b_1, \ldots, b_{n-1}$ computed as follows: $b_{n-1} = a_{n-1} - a_n$, $b_{n-2} = a_{n-2} - b_{n-1}$, ..., $b_1 = a_1 - b_2$, we say that r is equivalent to p if and only if $b_1 = v$.

Typed and Periodic Trees of Recurrences. Induction variable analysis in the context of typed operations is not new: all the compilers that have loop optimizers based on typed intermediate representations have solved this problem. However there is little literature that describes the problems and solutions [33]: these details are often considered too low level, and language dependent. As illustrated in the fifth introductory example, in Figure 1, the analysis of data dependences has to correctly handle the effects of overflowing on variables that are indexing the data. One of the solutions for preserving the semantics of wrapping types on TREC operations is to type the TREC and to map the effects of types from the SSA representation to the TREC representation. For example, the conversion from *unsigned char* to *unsigned int* of TREC $\{(uchar)100, +, (uchar)240\}_1$ is $\{(uint)100, +, (uint)\texttt{0xfffffff0}\}_1$, such that the original sequence remains unchanged $(100, 84, 68, \ldots)$. The first step of a TREC conversion proves that the sequence does not wrap. In the previous example, if the number of iterations in loop 1 is greater than 6, the converted TREC should also contain a wrap modulo 256, as illustrated by the first values of the sequence: $100, 84, 68, 52, 36, 20, 4, 244, 228, \ldots$. When it is impossible to prove that an evolution cannot wrap, it is safe to assume that it wraps, and keep the cast: $(uint)(\{(uchar)100, +, (uchar)240\}_1)$. Another solution is to use a periodic TREC, that lists all the values in a period: in the previous example we would have to store 15 values. Using periodic TREC for sequences wrapping over narrow types can seem practical, but this method is not practical for arbitrary sequences over wider types. Periodic sequences may also be generated by flip-flop operations, that are special cases of self referenced peeled TREC. Variables in a flip-flop exchange their initial values over the iterations, for example:

$$flip \to (3, 5, flip)_k(x) = [3, 5]_k(x) = \begin{cases} 3 & \text{if } x = 0 \bmod 2, \\ 5 & \text{if } x = 1 \bmod 2. \end{cases}$$

Exponential Trees of Recurrences. The exponential MCR [3] used by [32] and then extended to handle sums or products of polynomial and exponential evolutions [31] are useless in compiler technology for typed integer sequences, as integer typed arithmetic has limited domains of definition. Any overflowing operation either has defined modulo semantics, or is not defined by the language standard. The longer exponential integer sequence that can exist for an integer type of size 2^n is $n-1$: left shifting the first bit $n-2$ times. Storing exponential evolutions as peeled TREC seems efficient, because in general $n \leq 64$. We acknowledge that exponential MCR can have applications in compiler technology for floating point evolutions, but we have intentionally excluded floating point evolutions for simplifying this presentation. The next section will present our efficient algorithm that translates a part of the SSA dealing with scalar variables to TREC.

3 Analysis of Scalar Evolutions

We will now present an algorithm to compute closed-form expressions for inductive variables. Our algorithm translates a subset of the SSA to the TREC representation, interprets a part of the expressions and enriches the available information with properties that it computes, as the number of iterations, or the value of a variable at the end of a loop. It extends the applicability of classic optimizations, and allows the extraction of precise high level informations. We have designed our analyzer such that it does not assume a particular control-flow structure and makes no restriction on the recursive intricate variable definitions. It however fails to analyze irreducible control flow graphs [1], for which an uncomputable evolution $\top$ is returned. Our analysis does not use the syntactic information, making no distinction between names defined in the code or introduced by the compiler. The algorithm is also able to delay a part of the analysis

Fig. 3. Bird's eye view of the analyzer

until more information is known by leaving symbolic names in the target representation. The last constraint for inclusion in a production compiler is that the analyzer should be linear in time and space: even if the structure of our algorithm is complex, composed of a double recursion as sketched in Figure 3, it presents similarities with the algorithm for linear unification by Paterson and Wegman [24], where the double recursion is hidden behind a single recursion with a stack.

3.1 Algorithm

Figures 4 and 5 present our algorithm to compute the scalar evolutions of all variables defined by loop-ϕ nodes: ComputeLoopPhiEvolutions is a driver that

Algorithm: ComputeLoopPhiEvolutions
Input: SSA representation of the procedure
Output: a TREC for every variable defined by loop-ϕ nodes
  For each loop l in a depth-first traversal of the loop nest
    For each loop-ϕ node n in loop l
      InstantiateEvolution(AnalyzeEvolution(l, n), l)

Algorithm: AnalyzeEvolution(l, n)
Input: l the current loop, n the definition of an SSA name
Output: TREC for the variable defined by n within l
  `v` $\leftarrow$ variable defined by n
  $ln \leftarrow$ loop of n
  If Evolution[n] $\neq \bot$ Then $res \leftarrow$ Evolution[n]
  Else If n matches "`v = constant`" Then $res \leftarrow$ `constant`
  Else If n matches "`v = a`" Then $res \leftarrow$ AnalyzeEvolution(l, `a`)
  Else If n matches "`v = a` $\odot$ `b`" (with $\odot \in \{+, -, *\}$) Then
    $res \leftarrow$ AnalyzeEvolution(l, `a`) $\odot$ AnalyzeEvolution(l, `b`)
  Else If n matches "`v = loop-`ϕ`(a, b)`" Then
    (notice `a` is defined outside loop ln and `b` is defined in ln)
    Search in depth-first order a path from `b` to `v`:
    ($exist$, $update$) $\leftarrow$ BuildUpdateExpr(n, definition of `b`)
    If not $exist$ (if such a path does not exist) Then $res \leftarrow (\mathtt{a}, \mathtt{b})_l$: a peeled TREC
    Else If $update$ is $\top$ Then $res \leftarrow \top$
    Else $res \leftarrow \{\mathtt{a}, +, update\}_l$: a TREC
  Else If n matches "`v = condition-`ϕ`(a, b)`" Then
    $eva \leftarrow$ InstantiateEvolution(AnalyzeEvolution(l, `a`), ln)
    $evb \leftarrow$ InstantiateEvolution(AnalyzeEvolution(l, `b`), ln)
    If $eva = evb$ Then $res \leftarrow eva$ Else $res \leftarrow \top$
  Else $res \leftarrow \top$
  Evolution[n] $\leftarrow res$
  Return Eval(res, l)

Fig. 4. ComputeLoopPhiEvolutions and AnalyzeEvolution

illustrates the use of the analyzer and instantiation. In general, AnalyzeEvolution is called for a given loop number and a variable name. The evolution functions are stored in a database that is visible only to AnalyzeEvolution, and that is accessed using Evolution[n], for an SSA name n. The initial value for a not yet analyzed name is $\bot$. The cornerstone of the algorithm is the search and reconstruction of the symbolic update expression on a path of the SSA graph: BuildUpdateExpr. This corresponds to a depth-first search algorithm in the SSA graph with a pattern matching rule at each step, halting either with a success on the starting loop-ϕ node, or with a fail on any other loop-ϕ node of the same loop. Based on these results, AnalyzeEvolution constructs either a TREC or a peeled TREC. InstantiateEvolution substitutes symbolic parameters in a TREC. It computes their statically known value, i.e., a constant,

Algorithm: BuildUpdateExpr(h, n)
Input: h the halting loop-ϕ, n the definition of an SSA name
Output: (*exist*, *update*), *exist* is true if h has been reached,
update is the reconstructed expression for the overall effect in the loop of h
  If (n is h) Then Return (true, 0)
  Else If n is a statement in an outer loop Then Return (false, $\perp$),
  Else If n matches "`v = a`" Then Return BuildUpdateExpr(h, definition of `a`)
  Else If n matches "`v = a + b`" Then
    (*exist*, *update*) $\leftarrow$ BuildUpdateExpr(h, `a`)
    If *exist* Then Return (true, *update* + `b`),
    (*exist*, *update*) $\leftarrow$ BuildUpdateExpr(h, `b`)
    If *exist* Then Return (true, *update* + `a`)
  Else If n matches "`v = loop-`ϕ`(a, b)`" Then $ln \leftarrow$loop of n
    (notice `a` is defined outside ln and `b` is defined in ln)
    If `a` is defined outside the loop of h Then Return (false, $\perp$)
    $s \leftarrow$ Apply(ln, AnalyzeEvolution(ln, n), NumberOfIterations(ln))
    If s matches "`a + t`" Then (*exist*, *update*) $\leftarrow$ BuildUpdateExpr(h, `a`)
      If *exist* Then Return (*exist*, *update* + t)
  Else If n matches "`v = condition-`ϕ`(a, b)`" Then
    (*exist*, *update*) $\leftarrow$ BuildUpdateExpr(h, `a`)
    If *exist* Then Return (true, $\top$)
    (*exist*, *update*) $\leftarrow$ BuildUpdateExpr(h, `b`)
    If *exist* Then Return (true, $\top$)
  Else Return (false, $\perp$)

Algorithm: InstantiateEvolution(*trec*, l)
Input: *trec* a symbolic TREC, l the instantiation loop
Output: an instantiation of *trec*
  If *trec* is a constant `c` Then Return `c`
  Else If *trec* is a variable `v` Then
    If `v` has not been instantiated
      Mark `v` as instantiated and Return AnalyzeEvolution(l, `v`)
    Else `v` is in a mixer structure, Return $\top$
  Else If *trec* is of the form $\{e_1, +, e_2\}_x$ Then
    Return $\{$InstantiateEvolution$(e_1, l), +,$ InstantiateEvolution$(e_2, l)\}_x$
  Else If *trec* is of the form $(e_1, e_2)_x$ Then
    Return Unify((InstantiateEvolution(e_1, l), InstantiateEvolution$(e_2, l))_x$)
  Else Return $\top$

Fig. 5. BuildUpdateExpr and InstantiateEvolution algorithms

a periodic function, or an approximation with intervals, possibly triggering other computations of TREC in the process. The call to InstantiateEvolution is postponed until the end of the depth-first search, avoiding early approximations in the computation of update expressions. Combined with the introduction of symbolic parameters in the TREC, postponing the instantiation alleviates the need for a specific ordering of the computation steps. The correctness and complexity of this algorithm are established by structural induction [25].

3.2 Application of the Analyzer to an Example

We illustrate the analysis of scalar evolutions algorithm on the first introductory example in Figure 1, with the analysis of `c = φ(a, f)`. The SSA edge exiting the loop, Figure 6.(1), is left symbolic. The analyzer starts a depth-first search,

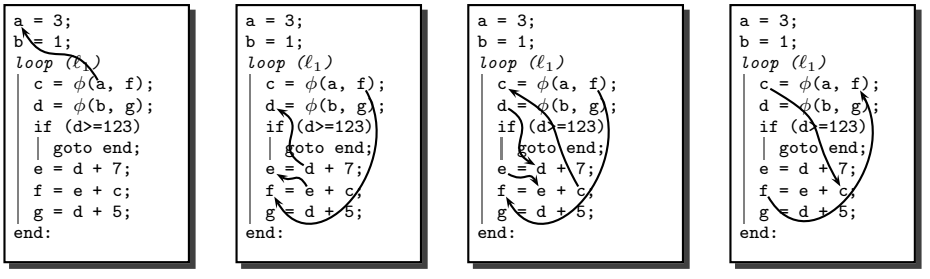


Fig. 6. Application to the first example

illustrated in Figure 6.(2): the edge `c`→`f` is followed to the definition `f = e + c`, then following the first edge `f`→`e`, reaches the assignment `e = d + 7`, and finally `e`→`d` leads to a loop-ϕ node of the same loop. Since this is not the starting loop-ϕ, the search continues on the other unexplored operands: back on `e = d + 7`, operand `7` is a scalar, then back on `f = e + c`, the edge `f`→`c` is followed to the starting loop-ϕ node, as illustrated in Figure 6.(3). Following the path of iterative updates in execution order, as illustrated in Figure 6.(4), gives the update expression: `e`. Finally, the analyzer records the TREC $\mathtt{c} = \{\mathtt{a}, +, \mathtt{e}\}_1$. An instantiation of $\{\mathtt{a}, +, \mathtt{e}\}_1$ yields: $\mathtt{a} = 3$, $\mathtt{e} = \{8, +, 5\}_1$, and $\{3, +, 8, +, 5\}_1$.

3.3 Applications

Empirical Study. To show the robustness and language-independence of our implementation, and to evaluate the accuracy of our algorithm, we determine a compact representation of *all* variables defined by loop-ϕ nodes in the SPEC CPU2000 [30] and JavaGrande [13] benchmarks. Figure 7 summarizes our experiments: affine univariate variables are very frequent because well structured loops are most of the time using simple constructs, affine multivariate less common, as they are used for indexing multi dimensional arrays. Difficult constructs such as polynomials of degree greater than one occur very rarely: we have detected only three occurrences in SPEC CPU2000, and none in JavaGrande. The last three columns in Figure 7 show the precision of the detector of the number of iterations: only the single-exit loops are exactly analyzed, excluding a big number of loops that contain irregular control flow (probably containing exception exits) as in the case of java programs. An approximation of the loop count can enable aggressive loop transformations as in the 171.swim SPEC CPU2000 benchmark, where the data accesses are used to determine a safe loop bound, allowing safe refinements of the data dependence relations.

Benchmark	U.	M.	C.	⊤	Loops	Trip	A.
CINT2000	12986	20	13526	52656	10593	1809	82
CFP2000	13139	52	12051	12797	6720	4137	68
JavaGrande	334	0	455	866	481	84	0

Fig. 7. Induction variables and loop trip count. Break-down of evolutions into: "U." affine univariate, "M." affine multivariate, "C." other compound expressions containing determined components, and "⊤" undetermined evolutions. Last columns describe: "Loops" the number of *natural* loops, "Trip" the number of *single-exit* loops whose trip count is successfully analyzed, "A." the number of loops for which an upper bound approximation of the trip count is available.

Optimization Passes. Based on our induction variable analysis, several scalar and high level loop optimizations have been contributed: Zdeněk Dvořák from SuSE has contributed strength reduction, induction variable canonicalization and elimination, and loop invariant code motion [1]. Dorit Naishlos from IBM Haifa has contributed a "simdization" pass [7,20] that rewrites loops to use SIMD instructions such as Altivec, SSE, etc. Daniel Berlin from IBM Research and Sebastian Pop have contributed a linear loop transformation framework [5] that enables the loop interchange transformation: on the 171.swim benchmark a critical loop is interchanged, giving 1320 points compared to 796 points without interchange: a 65% benefit. Finally, Diego Novillo from RedHat has contributed a value range propagation pass [22]. The dependence-based transformations use uniform dependence vectors [4], but our method for identifying conflicting accesses between TREC can be applicable to the computation of more general dependence abstractions, tests for periodic, polynomial, exponential or envelope TREC. We implemented an extended Banerjee test [4], and we will integrate the Omega test [28] in the next GCC version 4.2. In order to show the effectiveness of the Banerjee data dependence analyzer as used in an optimizer, we have measured the compilation time of the vectorization pass: for SPEC CPU2000 benchmarks, the vectorization pass does not exceed 1 second, nor 5 percent of the compilation time per file. The experiments were performed on a Pentium4 2.40 GHz with 512 Kb of cache, 2 GB of RAM, on a Linux kernel 2.6.8. Figure 8 illustrates the scalability and accuracy of the analysis: we computed all depen-

Benchmark	# tests	d	i	u	ZIV	SIV	MIV
CINT2000	303235	73180	105264	124791	168942	5301	5134
CFP2000	655055	47903	98682	508470	105429	17900	60543
JavaGrande v2.0	87139	13357	67366	6416	76254	2641	916

Fig. 8. Classification of data dependence tests in SPEC CPU2000 and JavaGrande. Columns "d", "i" and "u" represent the number of tests classified as dependent, independent, and undetermined. Last columns split the dependence tests into "ZIV", "SIV", "MIV": zero, single and multiple induction variable.

dences between pairs of references — both accessing the same array — in every function. We have to stress that this evaluation is quite artificial because an optimizer would focus the data dependence analysis only on a few loop nests. The number of MIV dependence tests witness the stress on the analyzer: these tests involve arrays accessed in different loops, that could be separated by an important number of statements. Even with these extreme test conditions, our data dependence analyzer catches an important number of dependence relations, and the worst case is 15 seconds and 70 percent of the compilation time.

4 Comparison with Closely Related Works

Induction variable detection has been studied extensively in the past because of its central role in loop optimizations. Our target closed form expressions is an extension of the chains of recurrences [3,32,31]. The starting representation is close to the one used in the Open64 compiler [8,16], but our algorithm avoids the syntactic case distinction made in [16] that has severe consequences in terms of generality (when analyzing intricate SSA graphs) and maintainability: as syntactic information is altered by several transformation passes, pattern matching at a low level may lead to an explosion of the number of cases to be recognized; e.g., if a simple recurrence is split across two variables, its evolution would be classified as wrap around if not handled correctly in a special case; in practice, [16] does not consider these cases. Path-sensitive approaches have been proposed [31,29] to increase precision in the context of conditional variable updates. These techniques may lead to an exponential number of paths, and although interesting, seem not yet suitable for a production compiler, where even quadratic space complexity is unacceptable on benchmarks like GNU Go[9].

Our work is based on the previous research results presented in [32]. We have experimented with similar algorithms and dealt with several restrictions and difficulties that remained unsolved in later papers: for example, loop sequences are not correctly handled, unless inserting at the end of each loop an assignment for each variable modified in the loop and then used after the loop. Because they are using a representation that is not in SSA form, they have to deal with all the difficulties of building an "SSA-like" form. With some minor changes, their algorithm can be seen as a translation from an unstructured list of instructions to a weak SSA form restricted to operations on scalars. This weak SSA form could be of interest for representations that cannot be translated to classic SSA form, as the RTL representation of GCC. Another interesting result for their algorithm would be a proof that constructing a weak SSA representation is faster than building the classic SSA representation, however they have not presented experimental results on real codes or standard benchmarks for showing the effectiveness of their approach. In contrast, our algorithm is analyzing a classic SSA representation, and instead of worrying about enriching the expressiveness of the intermediate representation, we are concerned about the opposite question: how to limit the expressiveness of the SSA representation in order to provide the optimizers a level of abstraction that they can process. It might well be argued that

a new representation is not necessary for concepts that can be expressed in the SSA representation: this point is well taken. We acknowledge that we could have presented the current algorithm as a transformer from SSA to an abstract SSA, containing abstract elements. However, we deliberately have chosen to present the analyzer producing trees of recurrences for highlighting the sources of our inspiration and for presenting the extensions that we proposed to the chains of recurrences. Finally, we wanted the algorithm presented in this paper to reflect the underlying implementation in GCC.

5 Conclusion and Perspectives

We introduced *trees of recurrences*, a formalism based on *multivariate chains of recurrences* [3,14], with symbolic and algebraic extensions, such as the peeled chains of recurrences. These extensions increase the expressiveness of standard chains of recurrences and alleviate the need to resort to intractable exponential expressions to handle wrap-around and mixer induction variables. We extended this representation with the evolution envelopes that handle abstract elements as approximations of runtime values. We also presented a novel algorithm for the analysis of scalar evolutions. This algorithm is capable of traversing an arbitrary program in Static Single-Assignment (SSA) form, without prior classification of the induction variables. The algorithm is proven by induction on the structure of the SSA graph. Unlike prior works, our method does not attempt to retrieve more complex closed form expressions, but focuses on generality: starting from a low-level three-address code representation that has been seriously scrambled by complex phases of data- and control-flow optimizations, the goal is to recognize simple and tractable induction variables whose algebraic properties allow precise static analysis, including accurate dependence testing. We have implemented and integrated our algorithm in a production compiler, the GNU Compiler Collection (4.0), showing the scalability and robustness of an implementation that is the basis for several optimizations being developed, including vectorization, loop transformations and modulo-scheduling. We presented experimental results on the SPEC CPU2000 and JavaGrande benchmarks, with an application to dependence analysis. Our results show no degradations in compilation time. Independently of the algorithmic and formal contributions to induction variable recognition, this work is part of an effort to bring competitive loop transformations to the free production compiler GCC.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.
2. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan and Kaufman, 2002.
3. O. Bachmann, P. S. Wang, and E. V. Zima. Chains of recurrences a method to expedite the evaluation of closed-form functions. In *Proceedings of the international symposium on Symbolic and algebraic computation.* ACM Press, 1994.

4. U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Boston, 1992.
5. D. Berlin, D. Edelsohn, and S. Pop. High-level loop optimizations for GCC. In *Proceedings of the 2004 GCC Developers Summit*, 2004. `http://www.gccsummit.org/2004`.
6. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.
7. A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 82–93. ACM Press, 2004.
8. M. P. Gerlek, E. Stoltz, and M. J. Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. on Programming Languages and Systems*, 17(1):85–122, Jan. 1995.
9. Gnu go. `http://www.gnu.org/software/gnugo/gnugo.html`.
10. P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT '91: Proceedings of the international joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (CAAP '91): vol 1*, pages 169–192, New York, NY, USA, 1991. Springer-Verlag New York, Inc.
11. M. Haghighat and C. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Programming Languages and Systems*, 18(4), July 1996.
12. L. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in LNCS. Springer-Verlag, 1993.
13. Java grande forum. `http://www.javagrande.org`.
14. V. Kislenkov, V. Mitrofanov, and E. Zima. Multidimensional chains of recurrences. In *Proceedings of the 1998 international symposium on symbolic and algebraic computation*, pages 199–206. ACM Press, 1998.
15. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *ACM Symp. on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.
16. S.-M. Liu, R. Lo, and F. Chow. Loop induction variable canonicalization in parallelizing compilers. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, page 228. IEEE Computer Society, 1996.
17. J. Merill. GENERIC and GIMPLE: a new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers Summit*, 2003. `http://www.gccsummit.org/2003`.
18. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.
19. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.
20. D. Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004. `http://www.gccsummit.org/2004`.
21. D. Novillo. Tree SSA - a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers Summit*, 2003. `http://www.gccsummit.org/2003`.
22. D. Novillo. A propagation engine for gcc. In *Proceedings of the 2005 GCC Developers Summit*, 2005. `http://www.gccsummit.org/2005`.

23. K. O'Brien, K. M. O'Brien, M. Hopkins, A. Shepherd, and R. Unrau. Xil and yil: the intermediate languages of tobey. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, pages 71–82, New York, NY, USA, 1995. ACM Press.
24. M. S. Paterson and M. N. Wegman. Linear unification. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186, New York, NY, USA, 1976. ACM Press.
25. S. Pop, P. Clauss, A. Cohen, V. Loechner, and G.-A. Silber. Fast recognition of scalar evolutions on three-address ssa code. Technical Report A/354/CRI, Centre de Recherche en Informatique (CRI), École des mines de Paris, 2004. `http://www.cri.ensmp.fr/classement/doc/A-354.ps`.
26. S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. Technical Report A/367/CRI, Centre de Recherche en Informatique (CRI), École des mines de Paris, 2005. `http://www.cri.ensmp.fr/classement/doc/A-367.ps`.
27. B. Pottenger and R. Eigenmann. Parallelization in the presence of generalized induction and reduction variables. In *ACM Int. Conf. on Supercomputing (ICS'95)*.
28. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, Aug. 1992.
29. S. Rus, D. Zhang, and L. Rauchwerger. The value evolution graph and its use in memory reference analysis. In *Proceedings of the 2004 Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004.
30. Standard performance evaluation corporation. `http://www.spec.org`.
31. R. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 106–115, 2004.
32. R. A. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'01)*, pages 118–132, 2001.
33. H. Warren. *Hacker's Delight*. Addison-Wesley, 2003.
34. M. J. Wolfe. Beyond induction variables. In *ACM Symp. on Programming Language Design and Implementation (PLDI'92)*, pages 162–174, San Francisco, California, June 1992.
35. M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
36. E. V. Zima. On computational properties of chains of recurrences. In *Proceedings of the 2001 international symposium on symbolic and algebraic computation*, pages 345–352. ACM Press, 2001.

Garbage Collection Hints

Dries Buytaert, Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere

ELIS Department, Ghent University – HiPEAC Member,
St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{dbuytaer, kvenster, leeckhou, kdb}@elis.UGent.be

Abstract. This paper shows that Appel-style garbage collectors often make suboptimal decisions both in terms of *when* and *how* to collect. We argue that garbage collection should be done when the amount of live bytes is low (in order to minimize the collection cost) and when the amount of dead objects is high (in order to maximize the available heap size after collection). In addition, we observe that Appel-style collectors sometimes trigger a nursery collection in cases where a full-heap collection would have been better.

Based on these observations, we propose *garbage collection hints (GCH)* which is a profile-directed method for guiding garbage collection. Offline profiling is used to identify favorable collection points in the program code. In those favorable collection points, the VM dynamically chooses between nursery and full-heap collections based on an analytical garbage collector cost-benefit model. By doing so, GCH guides the collector in terms of *when* and *how* to collect. Experimental results using the SPECjvm98 benchmarks and two generational garbage collectors show that substantial reductions can be obtained in garbage collection time (up to 30X) and that the overall execution time can be reduced by more than 10%.

1 Introduction

Garbage collection (GC) is an important subject of research as many of todays programming language systems employ automated memory management. Popular examples are Java, C# and .NET. Before discussing the contributions of this paper, we revisit some garbage collection background and terminology.

1.1 Garbage Collection

An Appel-style generational copying collector divides the heap into two generations [2], a variable-size *nursery* and a *mature generation*. Objects are allocated from the nursery. When the nursery fills up, a *nursery collection* is triggered and the surviving objects are copied into the mature generation. When the objects are copied, the size of the mature generation is grown and the size of the nursery is reduced accordingly. Because the nursery size decreases, the time between consecutive collections also decreases and objects have less time to die. When

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 233–248, 2005.

the nursery size drops below a given threshold, a *full-heap collection* is triggered. After a full-heap collection all free space is returned to the nursery.

In this paper we consider two flavors of generational copying collectors, namely *GenMS* and *GenCopy* from JMTk [3]. GenMS collects the mature generation using the mark-sweep garbage collection strategy. The GenCopy collector on the other hand, employs a semi-space strategy to manage its mature generation. The semi-space collector copies scanned objects, whereas the mark-sweep collector does not.

To partition the heap into generations, the collector has to keep track of references between different generations. Whenever an object in the nursery is assigned to an object in the mature generation—i.e. there is a reference from an object in the mature generation to an object in the nursery—this information is tracked by using a so-called *remembered set*. When a nursery collection is triggered the remembered set must be processed to avoid erroneously collecting nursery objects that are referenced only from the mature generation.

1.2 Paper Contributions

While garbage collection offers many benefits, the time spent reclaiming memory can account for a significant portion of the total execution time [1]. Although garbage collection research has been a hot research topic for many years, little research has been done to decide *when* and *how* garbage collectors should collect.

Garbage is typically collected when either the heap, or a generation is full. Ideally, the heap should be collected when the live ratio is low: the fewer live objects, the fewer objects need to be scanned and/or copied, the more memory there is to be reclaimed, and the longer we can postpone the next garbage collection run. In this paper, we show that collecting prior to a full heap, at points where the live ratio is low, can yield substantial reductions in GC time.

In addition, when using an Appel-style collector with two generations, a decision needs to be made whether to trigger a full-heap or nursery collection. We found that triggering nursery collections until the nursery size drops below a certain threshold is sometimes suboptimal. In this paper, we show how to trade off full-heap collections and nursery collections so that performance improves substantially.

The approach presented in this paper to decide *when* and *how* to collect, is called *garbage collection hints (GCH)* and works as follows. GCH first determines *favorable collection points (FCPs)* for a given application through offline profiling. A favorable collection point is a location in the application code where the cost of a collection is relatively cheap. During program execution a cost function is then computed in each FCP to determine the best GC strategy: postpone GC, perform a nursery GC, or perform a full-heap GC. Our experimental results using the SPECjvm98 benchmarks and two generational collectors show that GCH can reduce the garbage collector time by up to 30X and can improve the overall execution time by more than 10%.

Figure 1 perfectly illustrates why garbage collection hints actually work for the `_213_javac` benchmark. This graph shows the number of live bytes as a

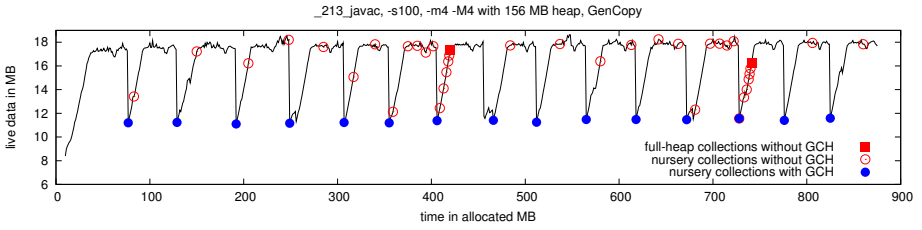


Fig. 1. Garbage collection points with and without GCH

function of the number of allocated bytes. The empty circles denote nursery collections and the squares denote full-heap collections when GCH is not enabled. Without GCH, GC is triggered at points where the number of live bytes is not necessarily low. In fact, the maximum GC time that we observed on our platform for these GC points is 225ms; and 12MB needs to be copied from the nursery to the mature generation. The GC time for a full-heap collection takes 330ms. When GCH is enabled (see the filled circles in Figure 1), garbage gets collected when the amount of live bytes reaches a minimum, i.e. at an FCP. The GC time at an FCP takes at most 4.5ms since only 126KB needs to be copied. From this example, we observe two key features why GCH actually works: (i) GCH preferably collects when the amount of live data on the heap is low, and (ii) GCH eliminates a number of full-heap collections by trading them for (cheaper) nursery collections.

The main contributions of this paper are as follows.

- We show that GC is usually not triggered when the amount of live data is low, i.e. when the amount of garbage collection work is minimal.
- We show that the collector does not always make the best decision when choosing between a nursery and a full-heap collection.
- We propose GCH which is a feedback-directed technique based on profile information that provides hints to the collector about *when* and *how* to collect. GCH tries to collect at FCPs when the amount of live data is minimal and dynamically chooses between nursery and full-heap collections. The end result is significant reductions in GC time and improved overall performance. GCH is especially beneficial for applications that exhibit a recurring phase behavior in the amount of live data allocated during program execution.

2 Experimental Setup

2.1 Java Virtual Machine

We use the Jikes Research Virtual Machine 2.3.2 (RVM) [4] on an AMD Athlon XP 1500+ at 1.3 GHz with a 256KB L2-cache, running Linux 2.6. Jikes RVM is a Java virtual machine written almost entirely in Java. Jikes RVM uses a compilation-only scheme for translating Java bytecodes to native machine instructions. For our experiments we use the *FastAdaptive* profile: all methods are

initially compiled using a baseline compiler, but sampling is used to determine which methods to recompile using an optimizing compiler.

Because Jikes RVM is written almost entirely in Java, internal objects such as those created during class loading or those created by the runtime compilers are allocated from the Java heap. Thus, unlike with conventional Java virtual machines the heap contains both application data as well as VM data. We found that there is at least 8MB of VM data that is quasi-immortal. The presence of VM data has to be taken into account when interpreting the results presented in the remainder of this work.

Jikes RVM's memory management toolkit (JMTk) [3] offers several GC schemes. While the techniques presented in this paper are generally applicable to various garbage collectors, we focus on the *GenMS* and *GenCopy* collectors. Both been used in Jikes RVM's production builds that are optimized for performance.

To get around a bug in Jikes RVM 2.3.2 we increased the maximum size of the remembered set to 256MB. In order to be able to model the shrinking/growing behavior of the heap accurately, we made one modification to the original RVM. We placed the remembered set outside the heap.

Performance is measured using the Hardware Performance Monitor (HPM) subsystem of Jikes RVM. HPM uses (i) the `perfctr`[1] Linux kernel patch, which provides a kernel module to access the processor hardware, and (ii) PAPI [5], a library to capture the processor's performance counters. The hardware performance counters keep track of the number of retired instructions, elapsed clock cycles, etc.

2.2 Benchmarks

To evaluate our mechanism, we use the SPECjvm98[2] benchmark suite. The SPECjvm98 benchmark suite is a client-side Java benchmark suite consisting of seven benchmarks, each with three input sets: `-s1`, `-s10` and `-s100`. With the `-m` and `-M` parameters the benchmark can be configured to run multiple times without stopping the VM. Garbage collection hints work well for long running applications that show some recurring phase behavior in the amount of live data. To mimic such workloads with SPECjvm98, we use the `-s100` input set in conjunction with running the benchmarks four times (`-m4 -M4`).

We used all SPECjvm98 benchmarks except one, namely `_222_mpegaudio`, because it merely allocates 15MB each run and triggers few GCs. The other benchmarks on the other hand, allocate a lot more.

All SPECjvm98 bencharmks are single-threaded except for `_227_mtrt` which is a multi-threaded raytracer. Note that because both Jikes RVM's sampling mechanism and the optimizing compiler run in separate threads the other benchmarks are not deterministic.

We ran all experiments with a range of different heap sizes. We vary the heap size between the minimum feasible heap size and the heap size at which our mechanism stops triggering or shows constant behavior. The larger the heap size, the less frequent garbage needs to be collected and the more time objects have to die.

[1] `http://user.it.uu.se/~mikpe/linux/perfctr/`
[2] `http://www.spec.org/jvm98/`

Some benchmarks like _213_javac use *forced garbage collections* triggered through calls to `java.lang.System.gc()`. For our experiments we disabled forced garbage collections unless stated otherwise.

3 Garbage Collection Hints

Our garbage collection hints approach consists of an offline and an online step, see Figure 2. The offline step breaks down into two parts: offline profiling of the application and garbage collector analysis. The offline profiling computes the live/time function of the application, i.e. the amount of live bytes as a function of the amount of bytes allocated. Based on this live/time function, favorable collection points (FCPs) can be determined. Determining the FCPs is a one-time cost per application. The garbage collector analysis characterizes the collection cost for a particular GC and application, i.e. the amount of time needed to process a given amount of live bytes. This is dependent on the collector and the platform on which the measurements are done. In the online part of GCH, the methods that have been identified as FCPs are instrumented to invoke a cost-benefit model that helps the garbage collector make decisions about *when* and *how* to collect. This decision making is based on the amount of heap space available, the live/time function of the application and the characteristics of the GC. The following subsections discuss GCH in more detail.

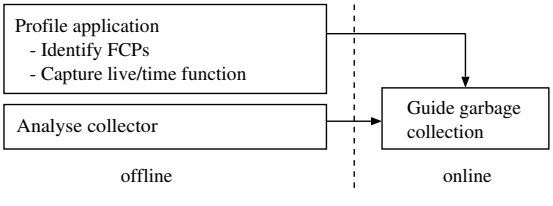


Fig. 2. An overview of the GCH methodology

3.1 Program Analysis

Live/Dead Ratio Behavior. The first step of the offline profiling is to collect the live/time function which quantifies the number of live bytes as a function of the bytes allocated so far. Moreover, we are interested in linking the live/time function to methods calls. We modified Jikes RVM to timestamp and report all method entries and exits. For each method invocation, we want to know how many objects/bytes died and how many objects are live. Therefore, a lifetime analysis is required at every point an object could have died. There are two reasons for an object to die: (i) an object's last reference is overwritten as a result of an assignment operation, or (ii) an object's last reference is on a stack frame and the stack frame gets popped because the frame's method returns or because an exception is thrown. To avoid having to do a lifetime analysis for

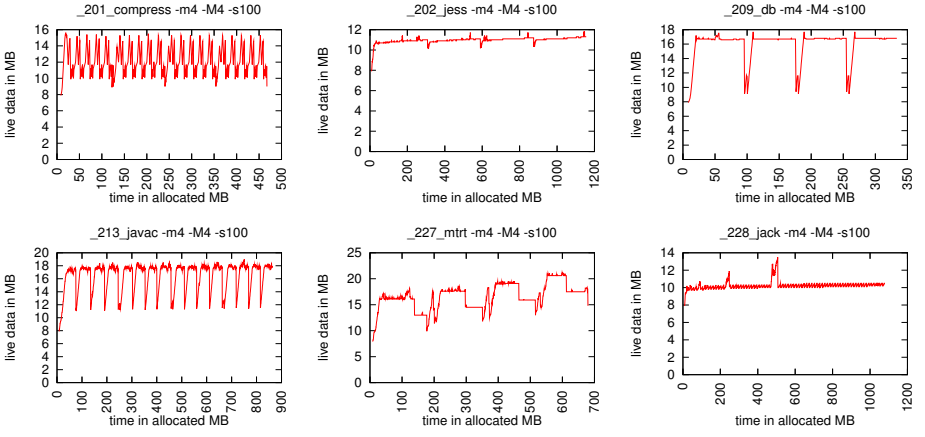


Fig. 3. The live/time function for the various benchmarks: number of live bytes as a function of the number of bytes allocated

every assignment operation, method return and exception, we used a modified version of the Merlin trace generator [6] that is part of Jikes RVM. Merlin is a tool that precisely computes every object's last reachable time. It has been modified to use our alternative timestamping method to correlate object death with method invocations.

Figure 3 shows the live/time function for the various benchmarks. As can be seen from these graphs, the live/time function shows recurring phase behavior. This recurring phase behavior will be exploited through GCH. Applications that do not exhibit a phased live/time function are not likely to benefit from GCH. Next, the live/time function is used to select FCPs and to compute the FCP live/time patterns.

Favorable Collection Points. For a method to represent a favorable collection point (FCP), it needs to satisfy three criteria:

1. An FCP's invocation should correspond to a local minimum in terms of the number of live bytes. In other words, we need to select methods that are executed in the minima of the live/time function. This will allow GCH to collect garbage with minimal effort.
2. An FCP should not be executed frequently. To minimize the overhead of the instrumentation code, FCPs that represent cold methods are preferred. A method that gets executed *only* in local minima is an ideal FCP.
3. The live/time pattern following the execution of the FCP should be fairly predictable, i.e. each time the FCP gets executed, the live/time function should have a more or less similar shape for some range after the FCP.

Given the live/time function, selecting FCPs is fairly straightforward. Table 1 shows the selected FCPs that we selected for the SPECjvm98 benchmarks. Some

Table 1. The selected FCPs for each of the benchmark applications

Benchmark	Favorable collection points
_201_compress	spec.io.FileInputStream.getContentLength()I
_202_jess	spec.benchmarks._202_jess.jess._undefrule.<init>()V
	spec.harness.BenchmarkTime.toString()Ljava/lang/String;
_209_db	spec.harness.Context.setBenchmarkRelPath(Ljava/lang/String;)V
	spec.io.FileInputStream.getCachingtime()J
_213_javac	spec.benchmarks._213_javac.ClassPath.<init>(Ljava/lang/String;)V
_227_mtrt	spec.io.TableOfExistingFiles.<init>()V
	spec.harness.Context.clearIOtime()V
	spec.io.FileInputStream.getCachingtime()J
_228_jack	spec.benchmarks._228_jack.Jack_the_Parser_Generator_Internals.-compare(Ljava/lang/String;Ljava/lang/String;)V

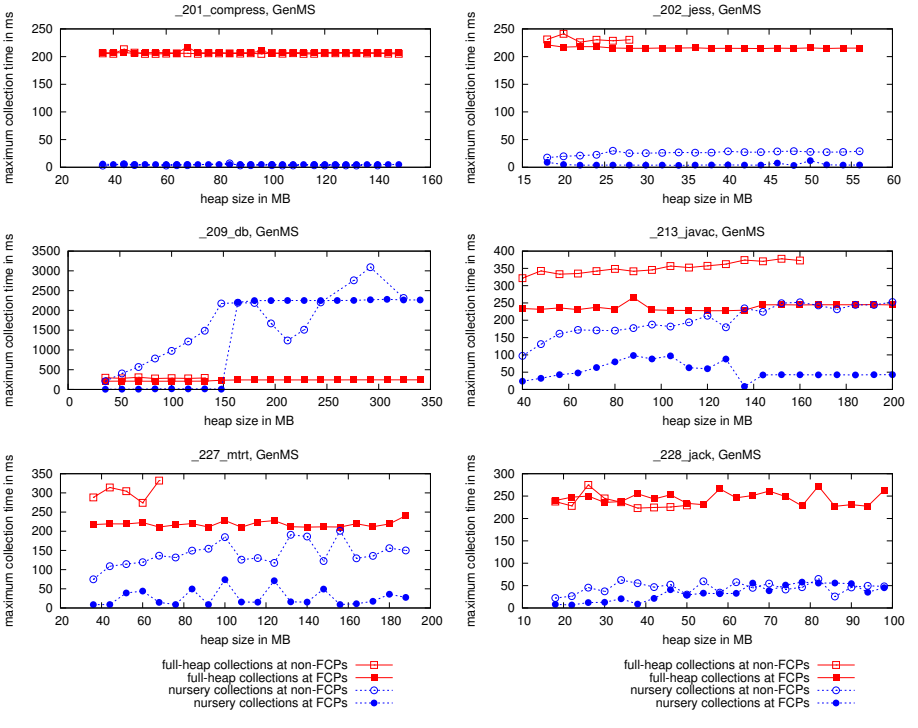


Fig. 4. The maximum garbage collection times across different heap sizes for each of the different scenarios

benchmarks have only one FCP (see for example Figure 1 for `_213_javac`); others such as `_227_mtrt` have three FCPs.

To illustrate the potential benefit of FCPs, Figure 4 plots the maximum time spent in GC when triggered at an FCP and when triggered in a non-FCP. We make a distinction between full-heap and nursery collections, and plot data for a range of heap sizes. For most benchmarks we observe that the maximum GC

time spent at an FCP is substantially lower than the GC time in a non-FCP. This reinforces our assumption that collecting at an FCP is cheaper than collecting in a non-FCP. However, there are two exceptions, `_201_compress` and `_228_jack`, for which GC time is insensitive to FCPs. For `_201_compress`, this is explained by the fact that the live/time function shown in Figure 3 is due to a few objects that are allocated in the Large Object Space (LOS). Because objects in LOS never get copied, GCH cannot reduce the copy cost. Furthermore, because there are only a few such objects it will not affect the scan cost either. For `_228_jack`, the height of the live/time function's peaks is very low, see Figure 3. Because `_201_compress` and `_228_jack` are insensitive to FCPs we exclude them from the other results that will be presented in this paper. (In fact, we applied GCH to these benchmarks but the result was a neutral impact of overall performance. Due to space constraints, we do not to include these benchmarks in the rest of this paper.)

It is also interesting to note that for `_209_db`, a nursery collection can be more costly than a full-heap collection. This is due to the fact that the remembered set needs to be scanned on a nursery collection. As such, for `_209_db` a full-heap collection can be more efficient than a nursery collection. This is exploited through GCH.

FCP's Live/Time Pattern. For each unique FCP we have to capture the *live/time pattern* following the FCP. This is a slice of the live/time function following the FCP that recurs throughout the complete program execution. We sample the FCP's live/time pattern at a frequency of one sample per 0.5MB of allocated memory and use it as input for the cost-benefit model. An FCP's live/time pattern is independent of the heap size (the same information is used for all heap sizes) and is independent of the collection scheme (the same information is used for both GenMS and GenCopy). It only needs to be computed once for each benchmark. For completeness, we add that in order to be independant of the collection scheme, we differentiate between data that is eligible to be copied and data that cannot be copied.

3.2 Collector Analysis

So far, we discussed the offline profiling that is required for GCH. We now discuss the characterization of the garbage collector. This characterization will be used in the cost model that will drive the decision making in GCH during program execution. The characterization of the garbage collector quantifies the cost of a collection. We identify three cost sources: the cost of a full-heap collection, the cost of a nursery collection and the cost of processing the remembered set. The cost functions take as input the amount of live data and output the estimated collection time. These cost functions are dependent on the application, the collector and the given platform (VM, microprocessor, etc.).

Figure 5 shows how the cost functions are to be determined for the GenMS and GenCopy collectors. The graphs are obtained by running the benchmarks multiple times with different heap sizes using instrumented collectors. In these

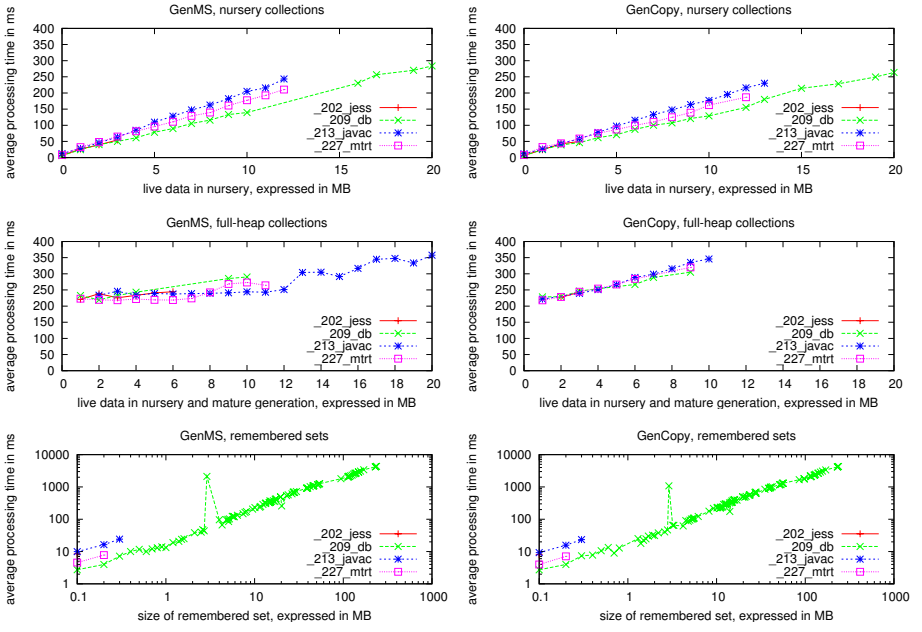


Fig. 5. The cost of a nursery and full-heap collection in terms of the amount of copied/live data

graphs we make a distinction between nursery collections, full-heap collections and processing of the remembered set. Hence, the processing times on the nursery collection graphs do not include the time required to process the remembered sets.

GC time appears to be a linear function of the amount of live data for both collectors. In other words, the scanning and copying cost is proportional to the amount of live bytes. Likewise, the processing cost of the remembered set appears to be a linear function of its size. In summary, we can compute linear functions that quantify the cost of a nursery collection, full-heap collection and processing of the remembered set.

In this paper we employ application-specific cost functions, i.e. we compute cost functions per application. In fact, on a specialized system with dedicated long running applications, it is appropriate to consider a cost function that is specifically tuned for the given application. Nevertheless, given the fact that the cost functions appear to be fairly similar across the various applications, see Figure 5, choosing application-independent cost functions could be a viable scenario for general-purpose environments. Such a scenario is likely to produce results that are only slightly worse compared to the ones presented in this paper.

3.3 GCH at Work

The information that is collected through our offline analysis is now communicated to the VM to be used at runtime. Jikes RVM reads all profile information

at startup. This contains (i) a list of methods that represent the FCPs, (ii) the live/time pattern per FCP, and (iii) the cost functions for the given garbage collector. Jikes RVM is also modified to dynamically instrument the FCPs. The instrumentation code added to the FCPs examines whether the current FCP should trigger a GC. The decision to collect or not is a difficult one as there exists a trade-off between reducing the amount of work per collection and having to collect more frequently. Clearly, triggering a collection will have an effect on subsequent collections. Because GC is invoked sooner due to GCH than without GCH, additional collections might get introduced. On the other hand, triggering a collection at an FCP can help reduce the GC overhead. A collection at an FCP will generally introduce modest pause times compared to collections at a non-FCPs. Moreover, triggering a full-heap collection grows the nursery size and gives objects more time to die, while triggering a nursery collection when few objects are live will result in the mature generation filling up slower, reducing the need for more expensive full-heap collections.

To make this complex trade-off, the instrumentation code in the FCPs implements an *analytical cost-benefit model*. The cost-benefit model estimates the total GC time for getting from the current FCP to the end of its FCP's live/time pattern. The cost-benefit model considers the following three scenarios: (i) do not trigger a GC in the current FCP, (ii) trigger a full-heap GC, or (iii) trigger a nursery GC. For each of these three scenarios, the cost-benefit model computes the total GC time by *analytically simulating* how the heap will evolve through the FCP's live/time pattern. This is done as follows. First, the cost-benefit model computes the GC cost *in the current FCP* under the three scenarios:

(i) The cost for not triggering a GC is obviously zero. The available heap size remains unchanged.
(ii) For computing the cost for triggering a full-heap collection in the current FCP, we first calculate the number of live bytes at the current FCP. We get this information from the live/time pattern. We subsequently use the full-heap GC cost function to compute the GC time given the amount of live data in the current FCP. The available heap size after the current (hypothetical) full-heap collection then equals the maximum heap size minus the amount of live data in the current FCP.
(iii) To compute the cost for triggering a nursery GC in the current FCP, we assume that the amount of live bytes in the nursery at that FCP is close to zero. The GC cost is then computed based on the nursery-GC cost function. This GC cost is incremented by an extra cost due to processing the remembered set. This extra cost is proportional to the size of the remembered set, which is known at runtime at an FCP. The heap size that was occupied by the nursery becomes available again for allocation.

In the second step of the cost-benefit model we compute the cost of additional collections over the FCP's live/time pattern for each of the three scenarios. In fact, for each scenario, the cost-benefit model analytically simulates how the heap will evolve over time when going through an FCP's live/time pattern. Therefore, we compute when the (nursery) heap will be full—when the

application has allocated all memory available in the heap. In case the system would normally trigger a full collection (i.e. when the nursery size drops below the given threshold), we need to compute the cost of a full-heap collection. This is done the same way as above, by getting the amount of live data from the FCP's live/time pattern—note that we linearly interpolate the live/time pattern—and use the major-GC cost function to compute its cost. In case the nursery size is above the given threshold, we need to compute the cost of a nursery collection. Computing the cost for a nursery collection is done by reading the number of live bytes from the FCP's live/time pattern and subtracting the number of live bytes in the previous GC point; this number gives us an estimate for the amount of live data in the nursery. This estimated amount of live nursery data is used through the nursery-GC cost function to compute an estimated nursery-GC cost.

When the end of the FCP's live/time pattern is reached within the model, an end cost is added to the aggregated GC cost calculated so far. This end cost is proportional to the fraction of used nursery space and indicates that the next (expected) collection will be close by or far away. After computing all the costs for each of the three scenarios, the scenario that results in the minimal total cost is chosen.

Note that the cost-benefit model presented above is specifically developed for GenMS and GenCopy, two Appel-style generational garbage collectors with a variable nursery size. However, a cost-benefit model could also be easily constructed for other collectors. For example, modifying this model for generational collectors with a fixed nursery, or non-generational collectors should not be that difficult.

3.4 GCH Across Inputs

GCH is a profile-driven garbage collection method which implies that the input used for the offline profiling run is typically different from the input used during online execution. Getting GCH to work across inputs needs a few enhancements since the size and the height of an FCP's live/time pattern varies across inputs; the general shape of the FCP's live/time pattern however is relatively insensistive to the given input. We define the height of an FCP's live/time pattern as the difference in live data at the top of an FCP's live/time pattern and at the FCP itself. For example, for `_213_javac`, see Figure 1, the height is approximately 6MB. The size of an FCP's live/time pattern is defined as the number of allocated bytes at an FCP's live/time pattern; this is approximately 60MB for `_213_javac`, see also Figure 1. To address this cross-input issue we just scale the size and the height of the FCP live/time pattern. In practice, the amount of live data at the top of an FCP's live/time pattern can be computed at runtime when a GC is triggered at the top of an FCP's live/time pattern. The amount of allocated bytes over an FCP's live/time pattern can be computed at run-time as well by just querying the VM. These scaling factors are then to be used to rescale the data in the FCP live/time pattern.

4 Experimental Results

4.1 Garbage Collection Time

In order to evaluate the applicability of GCH, we have set up the following experiment. We used the profile information from the `-s10` run to drive the execution of the `-s100` run after cross-input rescaling as discussed in section 3.4.

Figure 6 shows the reduction through GCH in GC time over a range of heap sizes. (Reduction is defined as $100 \times (1 - \frac{time_{GCH}}{time_{old}})$. Thus, a 50% reduction means GC time is halved.) Each reduction number is an average number over three runs; numbers are shown for both GenMS and GenCopy. Figure shows that GCH improves GC time for both collectors and for nearly all heap sizes. For both collectors GCH achieves substantial speedups, up to 30X for `_213_javac` and 8X for `_209_db`.

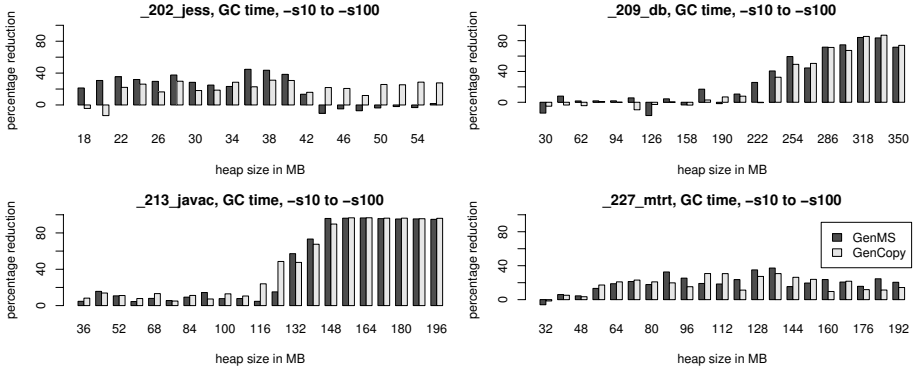


Fig. 6. Reduction in garbage collection time through GCH across inputs. The profile input is `-s10`; the reported results are for `-s100`.

The sources for these huge speedups are twofold. First, GCH generally results in fewer (nursery) collections than without GCH, see Table 2 which shows the average number of GCs over all heap sizes; we observe fewer GCs with GCH for all benchmarks except one, namely `_209_db` for which the number of collections remains unchanged with and without GCH (we will discuss `_209_db` later on). For `_213_javac` we observe a 30% reduction in the number of (nursery) collections. The second reason for these huge GC time speedups, is the substantial reduction in the amount of time spent per GC. This was already mentioned in Figure 4.

Note that also for `_209_db`, GCH is capable of reducing the GC time substantially, especially for large heap sizes. The reason is not the reduced number of collections, but the intelligent selection of full-heap collections instead of nursery collections. The underlying reason is that `_209_db` suffers from a very large remembered set. GCH triggers more full-heap collections that do not suffer from processing the remembered set, see Table 2. A full-heap collection typically only takes 250ms whereas a nursery collection can take up to 2,000ms, see Figure 4.

Table 2. The average number of garbage collections across all heap sizes with and without GCH

	GenMS collector				GenCopy collector			
	Full-heap		Nursery		Full-heap		Nursery	
Benchmark	no GCH	GCH	no GCH	GCH	no GCH	GCH	no GCH	GCH
_202_jess	0	1	245	186	2	3	349	294
_209_db	1	3	16	14	2	4	25	25
_213_javac	2	2	80	62	6	5	93	61
_227_mtrt	0	1	45	36	2	2	81	67

Note that the remembered set increases with larger heap sizes which explains the increased speedup for larger heap sizes. The two negative speedup results for the GenCopy collector for a 110MB and 140MB heap are due to particularities in the time-varying behavior of the remembered set in _209_db. Our general cost model of the remembered set works fairly well across all programs but does not cover all pecularities observed in _209_db. However, through an experiment we were able to tweak the cost model of the remembered set to _209_db which results in an overall GC time speedup (not included here due to space constraints). While the large remembered sets themselves are the consequence of the fact that JMTk uses sequential store buffers without a space cap, it shows that our analytic framework is robust in the face of extreme cases like this. This does not affect the generality of our approach since in many practical situations the cost model can be designed for a given application in a specialized environment.

Remind that because of the way Jikes RVM works, the heap contains both application data and VM data. We believe that our technique would be even more effective in a system where the collector does not have to trace VM data. In such a system, full-heap collections would generally be cheaper opening up extra opportunities to replace nursery collections by full-heap collections.

4.2 Overall Execution Time

Figure 7 depicts the impact of GCH on the overall execution time. For some benchmarks and heap sizes, the total execution time is more or less unaffected through GCH because the time spent collecting garbage is only a small fraction of the total execution time. This is the case for _227_mtrt. The small slowdowns or speedups observed are probably due to changed data locality behavior because of the changed GCs. However, for _202_jess, _213_javac and _209_db performance improved up to 5.7%, 12.5% and 10.5% respectively. For these benchmarks, the huge GC time speedups translate themselves in overall performance speedup.

4.3 Run-Time System Overhead

To explore the run-time overhead of our system, we compare the performance of a without-GCH Jikes RVM versus a with-GCH Jikes RVM. In the with-GCH version, the profile information is read, the FCPs are instrumented and at each

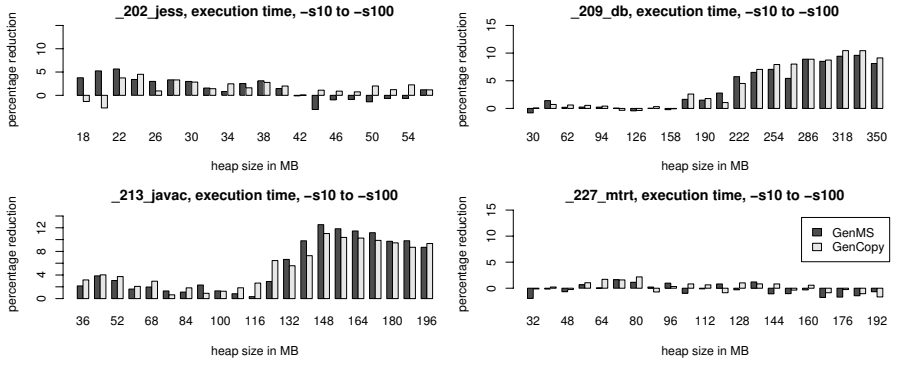


Fig. 7. Performance improvement in total execution time through GCH across inputs. The profile input is `-s10`; the reported results are for `-s100`.

invocation of an FCP the cost-benefit model is computed, however, it will never trigger a collection. For computing the overhead per benchmark, each benchmark is run multiple times and the average overhead is computed over these runs. The average overhead over all benchmarks is 0.3%; the maximum overhead is 1.3% for `_227_mtrt`.

5 Related Work

We now discuss previously proposed GC strategies that are somehow related to GCH, i.e. all these approaches implement a mechanism to decide when *or* how to collect. The work presented in this paper differs from previous work in that we combine the decision of both when *and* how to collect in a single framework.

The Boehm-Demers-Weiser (BDW) [7] garbage collector and memory allocator include a mechanism that determines whether to collect garbage or to grow the heap. The decision whether to collect or grow the heap is based on a static variable called the *free space divisor (FSD)*. If the amount of heap space allocated since the last garbage collection exceeds the heap size divided by FSD, garbage is collected. If not, the heap is grown. Brecht *et al.* [8] extended the BDW collector by taking into account the amount of physical memory available and by proposing dynamically varying thresholds for triggering collections and heap growths.

Wilson *et al.* [9] observe that (interactive) programs have phases of operation that are compute-bound. They suggest that tagging garbage collection onto the end of larger computational pauses, will not make those pauses significantly more disruptive. While the main goal of their work is to avoid or mitigate disruptive pauses, they reason that at these points, live data is likely to be relatively small since objects representing intermediate results of the previous computations have become garbage. They refer to this mechanism as *scavenge scheduling* but present no results.

More recently, Ding *et al.* [10] presented preliminary results of a garbage collection scheme called *preventive memory management* that also aims to exploit

phase behavior. They unconditionally force a garbage collection at the beginning of certain execution phases. In addition, they avoid garbage collections in the middle of a phase by growing the heap size unless the heap size reaches the hard upperbound of the available memory. They evaluated their idea using a single Lisp program and measured performance improvements up to 44%.

Detlefs *et al.* [11] present the garbage-first garbage collectors which aims at satisfying soft real-time constraints. Their goal is to spend no more than x ms during garbage collection for each y ms interval. This is done by using a collector that uses many small spaces and a *concurrent marker* that keeps track of the amount of live data per space. The regions containing most garbage are then collected first. In addition, collection can be delayed in their system if they risk violating the real-time goal.

Velasco *et al.* [12] propose a mechanism that dynamically tunes the size of the copy reserve of an Appel collector [2]. Tuning the copy reserve's size is done based on the ratio of surviving objects after garbage collection. Their technique achieves performance improvements of up to 7%.

Stefanovic *et al.* [13] evaluate the older-first generational garbage collector which only copies the oldest objects in the nursery to the mature generation. The youngest objects are not copied yet; they are given enough time to die in the nursery. This could be viewed of as a way deciding when to collect.

Recent work [14,15,16] selects the most appropriate garbage collector during program execution out of a set of available garbage collectors. As such, the garbage collector is made adaptive to the program's dynamic execution behavior. The way GCH triggers nursery or full-heap collections could be viewed as a special form of what these papers proposed.

6 Summary and Future Work

This paper presented garbage collection hints which is a profile-directed approach to guide garbage collection. The goal of GCH is to guide in terms of *when* and *how* to collect. GCH uses offline profiling to identify favorable collection points in the program code where the amount of live data is relatively small (in order to reduce the amount of work per collection) and the amount of dead bytes is relatively large (in order to increase the amount of available heap after collection). Triggering collections in these FCPs can reduce the number of collections as well as the amount of work per collection. Next to guiding when to collect, GCH also uses an analytical cost-benefit model to decide how to collect, i.e. whether to trigger a nursery or a full-heap collection. This decision is made based on the available heap size, and the cost for nursery and full-heap collections. Our experimental results using SPECjvm98 showed substantial reductions in GC time (up to 30X) and significant overall performance improvements (more than 10%).

In future work, we plan to extend and evaluate GCH for other collectors than the ones considered here. We also plan to study dynamically inserted garbage collection hints in which profiling is done online during program execution.

Acknowledgments

Dries Buytaert is supported by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Kris Venstermans is supported by a BOF grant from Ghent University, Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (FWO—Vlaanderen). We thank the reviewers for their insightful comments.

References

1. Blackburn, S.M., Cheng, P., McKinely, K.S.: Myths and realities: the performance impact of garbage collection. In: Proceedings of SIGMETRICS'04, ACM (2004)
2. Appel, A.W.: Simple generational garbage collection and fast allocation. Software practices and experience **19** (1989) 171–183
3. Blackburn, S.M., Cheng, P., McKinley., K.S.: Oil and water? High performance garbage collection in Java with JMTk. In: Proceedings of ICSE'04. (2004) 137–146
4. Alpern, B., Attanasio, C.R., Barton, J.J., Burke, M.G., Cheng, P., Choi, J.D., Cocchi, A., Fink, S.J., Grove, D., Hind, M., Hummel, S.F., Lieber, D., Litvinov, V., Mergen, M.F., Ngo, T., Russell, J.R., Sarkar, V., Serrano, M.J., Shepherd, J.C., Smith, S.E., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeño Virtual Machine. IBM Systems Journal **39** (2000) 211–238
5. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. The international journal of high performance computing applications **14** (2000) 189–204
6. Hertz, M., Blackburn, S.M., Moss, J.E.B., McKinley, K.S., Stefanovic, D.: Error free garbage collection traces: how to cheat and not get caught. In: Proceedings of SIGMETRICS'02, ACM (2002) 140–151
7. Boehm, H., Weiser, M.: Garbage collection in an uncooperative environment. Software practices and experience **18** (1988) 807–820
8. Brecht, T., Arjomandi, E., Li, C., Pham, H.: Controlling garbage collection and heap growth to reduce the execution time of Java applications. In: Proceedings of OOPSLA'01, ACM (2001) 353–366
9. Wilson, P.R., Moher, T.G.: Design of the opportunistic garbage collector. In: Proceedings of OOPSLA'89, ACM (1989) 23–35
10. Ding, C., Zhang, C., Shen, X., Ogihara, M.: Gated memory control for memory monitoring, leak detection and garbage collection. In: Proceedings of MSP'05, ACM (2005) 62–67
11. Detlefs, D., Flood, C., Heller, S., Printezis, T.: Garbage-first garbage collection. In: Proceedings of ISMM'04, ACM (2004) 37–48
12. J. M. Velasco, K. Olcoz, F.T.: Adaptive tuning of reserved space in an Appel collector. In: Proceedings of ECOOP'04, ACM (2004) 543–559
13. Stefanovic, D., Hertz, M., Blackburn, S.M., McKinley, K.S., Moss, J.E.B.: Older-first garbage collection in practice: evaluation in a Java virtual machine. In: Proceedings of MSP'02, ACM (2002) 25–36
14. Andreasson, E., Hoffmann, F., Lindholm, O.: Memory management through machine learning: to collect or not to collect? In: Proceedings of JVM'02, USENIX (2002)
15. Printezis, T.: Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment. In: Proceedings of JVM'01, USENIX (2001)
16. Soman, S., Krintz, C., Bacon, D.F.: Dynamic selection of application-specific garbage collectors. In: Proceedings of ISMM'04, ACM (2004) 49–60

Exploiting a Computation Reuse Cache to Reduce Energy in Network Processors

Bengu Li[1], Ganesh Venkatesh[2], Brad Calder[2], and Rajiv Gupta[1]

[1] University of Arizona, CS Dept., Tucson, AZ 85737
[2] University of California, San Diego, CSE Dept., La Jolla, CA 92093

Abstract. High end routers are targeted at providing worst case throughput guarantees over latency. Caches on the other hand are meant to help latency not throughput in a traditional processor, and provide no additional throughput for a balanced network processor design. This is why most high end routers do not use caches for their data plane algorithms.

In this paper we examine how to use a cache for a balanced high bandwidth network processor. We focus on using a cache not as a latency saving mechanism, but as an energy saving device. We propose using a *Computation Reuse Cache* that caches the answer to a query for data-plane algorithms, where the tags are the inputs to the query and the block the result of the query. This allows the data-plane algorithm to perform a complete query in one cache access if there is a hit. This creates slack by reducing the number of instructions executed. We then exploit this slack by fetch-gating the data-plane algorithm while matching the worst case throughput guarantees of the rest of the network processor. We evaluate the computation reuse cache for network data-plane algorithms IP-lookup, Packet Classification and NAT protocol.

1 Introduction

Network processors are designed to achieve high throughput. With the tremendous increase in line-speed, the amount of throughput required by network processors is increasing significantly. For example, OC-192 (10 Gig/Sec) requires a packet to be completed every 52 nanoseconds and OC-768 (40 Gig/Sec) has a packet completed every 13 nanoseconds.

In these network processors most of the packet processing algorithms are based on some utility data structures. The existence of these utility data structures is ubiquitous. These data structures involved in routing tasks are stored on-chip in large SRAMs for high end routers with latencies in the range of 30 cycles. Thus, much of the time spent during packet processing tasks is taken up by reading (and sometimes writing) of utility data structures in this SRAM. The three examples we focus on in this paper are IP-lookup, Packet Classification, and NAT protocol. In IP-lookup, a routing table is used to do longest prefix matching lookup [10,24]. In packet classification, a classification table is used to record the flow status [13,12]. In NAT protocol, maps are maintained so that

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 251–265, 2005.

source IP addresses, TCP ports, destination IP addresses and TCP ports are used to retrieve a translated source port [11].

In this paper we propose using a *Computation Reuse Cache* that caches the answer to a query for these data-plane algorithms, where the tags are the inputs to the query and the cache block the result of the query. This allows the data-plane algorithm to perform a complete query in one cache access if there is a hit. The computation reuse cache cannot increase the throughput of a balanced network processor. It is instead used to save energy. A hit in the computation reuse cache creates slack by reducing the number of instructions executed for the processing of a packet. This allows the processor to exploit this slack through fetch-gating for the data-plane algorithm while still matching the worst case throughput guarantees of the rest of the network processor. The use of the computation reuse cache is controlled by the data plane algorithm through a programmable interface consisting of specific instructions to lookup and use the cache. Therefore, how it is used (what the tag and data represent) can be different from one data plane algorithm to the next.

In Section 3 we describe why a traditional cache will not help a balanced network processor design. Section 4 examines how much value locality and temporal data locality exists in the fields of packet headers across packet streams in the network traffic for the three data-plane algorithms examined. Section 5 describes the design and use of our computation reuse cache, and Section 6 describes how to use it in combination with fetch gating to save energy. Section 7 provides performance results for our approach and we conclude in Section 8.

2 Related Work

We first briefly summarize related work on temporal locality and caching in network processors, reuse caching, and fetch gating.

2.1 Locality and Caching in Network Processors

Memik et al. [20] examine the use of a traditional cache for a set of networking applications on a StrongARM 110 processor. They found that most of the cache misses came from a small number of instructions. To address this, they use a filter for their data cache to remove the memory accesses with low locality. Li et al. [16] investigate a range of memory architectures that can be used for a wide range of packet classification caches. They study the impact of factors like cache associativity, cache size, replacement policy and the complexity of hash functions on the overall system performance. Both of these studies use a traditional cache, which cannot be used to increase the throughput of data plane algorithms for a balanced network processor. This is why our focus is on saving energy by using a computation reuse cache to create slack in the data-plane algorithm's schedule.

Chiueh et al. [6] use a combined hardware/software approach to construct a cache for high performance IP routing in general-purpose CPU based routers. The destination host address is mapped to a virtual address space and used to

lookup a destination route in the Host Address Cache (HAC). A part of the normal L1 data cache is reserved for use as the HAC. We would classify this approach as being similar to our computation reuse cache in that a hit in the cache skips the full IP lookup. In case of a lookup miss, a 3-level routing table lookup algorithm is consulted for the final routing decision. Their focus is on using the HAC to provide throughput for their network processor design, and not for energy savings. The contribution of our work is to define the general notion of using a programmable computation reuse cache for data plane algorithms and to use it to save energy in a balanced network processor while still providing worst-case throughput guarantees.

2.2 Instruction Reuse

Sodini and Sohi observe that many instructions or groups of instructions have the same input and generate the same output when dynamically executed. They exploit this phenomenon by buffering the computation result of the previous execution of instruction dependency chains. They use the same result for future dynamic instances of the same dependency chains if the input to these chains are the same. In this way, the execution of many groups of instructions can be avoided and the early outcome can allow dependent instructions to proceed sooner. They use a hardware mechanism called the Reuse Buffer (RB) to store the previous computation results. The program counter is used as an index to search the RB for cached chains. Our computation reuse cache is motivated by the reuse buffer, but it differs in that it is programmable and used to cache computations at much larger levels (methods) than just dependency chains.

Ding and Li [9] present a pure compiler memoization technique to exploit value locality. They detect code segments that are executed repeatedly, which generate a small number of different values. The code segment is replaced by a table recording the previous computation results for later lookup if the same values are seen. Performance improvement and energy consumption reduction are achieved. Their work is related to ours because we are also using a computation reuse mechanism to reduce energy consumption. Their approach is a purely software technique. In contrast, we provide a programmable hardware technique specific for data-plane algorithms in order to save energy while providing worst-case throughput guarantees.

2.3 Clock and Fetch Gating

Luo et al. [18] use a clock gating technique to reduce power consumption in multi-core network processors. They observe that when the incoming traffic rate is low, most processing elements on the network processor are nearly idle and yet still consume dynamic power. When the number of idle threads increase, they start to gate off the clock network of a processing unit. When the pressure from the incoming buffer rises, they stop clock gating. Since the activation takes time, they need extra buffer space to avoid packet loss. They also developed strategies to terminate and reschedule threads during activation and deactivation. This

work is related to ours because we both aim at gating some part of the network processor to reduce energy. We both use the queue occupancy information in gating decisions. Their approach is complementary to ours and can be used in tandem. Their focus is on applying fetch gating when the traffic rate is low, whereas we focus on applying fetch gating when we can find and exploit value locality, and this includes when the traffic rate is high.

Manne et al. [19] observe that due to branch mispredictions wrong path instructions cause a large amount of unnecessary work in wide-issue super-scalar processors. They develop a hardware mechanism called pipeline gating to control rampant speculation in the pipeline. They use a confidence estimator to assess the quality of each branch prediction. In case of low confidence, they gate the pipeline by stalling instruction fetch. Baniasadi and Moshovos [2] extend this approach to throttle the execution of instruction flow in wide-issue super-scalar processors to achieve energy reduction. They use instruction flow information such as rate of instructions passing through stages to determine whether to stall stages. When the rate is sufficiently high and there is enough instruction level parallelism, they may stall fetch because introducing extra instructions would not significantly improve performance. Karkhanis et al. [15] also propose a mechanism called Just-In-Time instruction delivery to save energy. They observe that performance-driven design philosophy causes useful instructions to be fetched earlier than needed and stall in the pipeline for many cycles or they wait in the issue queue. Also when a branch misprediction occurs, all those early-issued instruction along mispredicted branch are flushed. This wastes energy. Their suggested mechanism monitors and dynamically adjusts the maximum number of in-flight instructions in the processor according to processor performance. When a maximum number is reached, the instruction fetching is gated. Buyuktosunoglu et al. [5] collect issue queue statistics to resize the issue queue dynamically to improve issue queue energy and performance on a super-scalar processor. The statistics are derived from counters that keep track of the active state of each queue entry on a cycle-by-cycle basis. They divide the issue queue into separate chunks and may turn off/on certain block based on statistics. The above prior works are related to our work since they all gate/throttle the execution of instruction flow to achieve the goal of energy reduction. For our approach, we build upon these techniques to gate fetch for our data-plane micro-engines.

3 Why a Traditional Cache Does Not Help the Throughput of a Balanced Network Processor

High end routers are targeted at providing worst-case throughput guarantees over latency. A network processor that is a balanced design cannot have the throughput of its data-plane algorithms (e.g., ip-lookup, classification, etc) increased by adding a traditional cache. By traditional cache, we mean a cache that uses a standard memory address as its index and tag, and then a N-bytes of consecutive memory is stored in the cache block starting at the block address. A traditional cache being used for a data-plane algorithm would be indexed by

a standard load memory address to access an arbitrary level of the data-plane's algorithm's data structure. By *balanced design* we mean a network processor design where the memory latency is already completely hidden for the desired worst case throughput [22] and each of the components of the network processor are designed for the same worst case throughput guarantees. Balanced network processors are designed with enough overlapped execution (threads) that the latency to perform each memory lookup in the data-plane algorithm is completely hidden. In such a design, traditional caches cannot help increase throughput, since they just attack latency.

4 Value Locality in Network Processing

By value locality we refer to the phenomenon that when a stream of packets is examined, certain fields of the packet headers have the same values occurring in them repeatedly. The occurrence of value locality in the fields of packet headers are a direct result of the behavior of high level network protocols and network applications. For example, in a file transfer protocol, bursts of packets are generated between a specific pair of hosts within a short amount of time. Thus, the source and destination address fields exhibit value locality. The values in packet headers often directly determine which portions of the data-plane algorithm's data structure a network processing application will access. Therefore value locality in packet header fields gives rise to temporal locality in accesses to data in these data structures. In this section we examine how value locality in packet header fields result in temporal locality in accesses to data structures used in the three network processing tasks examined in this paper.

IP Routing Table Lookup. We first examine the longest prefix matching routing lookup application. The routing table is a set of entries each containing a destination network address, a network mask, and an output port identifier. Given a destination IP address, routing lookup uses the network mask of an entry to select the most significant bits from the destination address. If the result matches the destination network address of the entry, the output port identifier in the entry is a potential lookup result. Among such matches, the entry with the longest mask is the final lookup result [10]. In some implementations, the routing table is often organized as a trie, either multi-bit Patricia trie [24] or reduced radix tree [10]. Each lookup generates a sequence of accesses to the trie entries from the top of the trie down to the external nodes of the trie. Most important of all, the entries of the trie that are accessed are actually determined by the value of the destination IP address field of the packets. As shown by our study as well as work of other researchers [7,21], the destination IP address field shows high value locality. The routing lookup generates many sequences of memory accesses to the same trie entries over a fairly long interval of time and thus exhibiting temporal locality. We can exploit this temporal locality by putting the recently accessed trie entries with the routing lookup result in a computation reuse cache. Therefore other accesses in the near future, with the same destination, can then be avoided by checking this cache first.

Packet Classification Table Lookup. Packet classification is an essential part of network processing applications. Many applications require packet classification to classify packets into different flows based upon multiple fields from packet headers [13,12] (e.g., DiffServ uses a 5-tuple lookup consisting of IP source and destination addresses, the TCP source and destination ports, and the protocol field for classification). A classification table is used to record different flows which must be kept up to date. The entries accessed are actually determined by the above mentioned 5-tuple. Since once a flow is established, large number of packets in the flow are typically sent in a burst, the combination of this 5-tuple of fields demonstrates high value locality. The same classification table entry is accessed many times. We can exploit this behavior by caching the flow identification found so that closely following accesses to the same classification table entry can be avoided. Factors that affect packet classification are studied in [4,16].

NAT Portmapping Table Lookup. The Network Address Translation (NAT) protocol [11] can multiplex traffic from an internal network and present it to the Internet as if it was coming from a single computer having only one IP address in order to overcome the shortage of IP addresses, address security needs, or to ease and increase the flexibility of network administration. The NAT gateway uses a port mapping table that relates the client's real local IP address and its source port plus the translated source port number to a destination address and port. When a packet from an internal network is being sent out, the NAT gateway looks up the port mapping table and modifies the source address of the packet to the unique local network IP address and replaces the source port with translated source port. When a reply from the remote machine arrives, the port mapping table is used to reverse the procedure. The four fields in the internal network packet header, including the source address, source port, destination address and destination port, which defines a TCP/IP connection, determine which entry in the port mapping table is accessed. Since once a TCP/IP connection is established, bursts of packets in the connection will be sent out, these four fields have high value locality. As a result, the same entry in the port mapping table will be accessed very frequently and demonstrate temporal locality. We can exploit this behavior by caching the recently accessed entries in the port mapping table to avoid the lookups.

Table 1 summarizes the behavior of the applications discussed. Next we present results measuring the degree of value locality in these applications. For this study we use traces of IP packets taken from Auckland-II Trace Archive [1]. Characteristics of these traces, including the total number of packets and distinct destination addresses, are given in Table 2. We selected packet traces from the trace archive that had a large number of distinct destinations with respect to the number of packets in the trace.

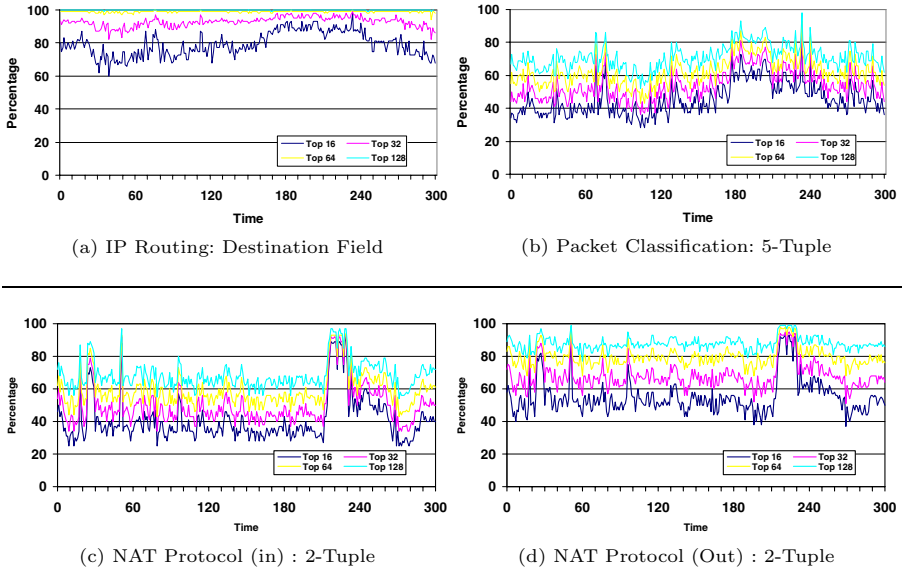
We measured the value locality in terms of the percentage of the packets which have a frequently occurring combination of values for the relevant n-tuple of header fields where n is the number of relevant header fields for a given application. According to Table 1, *IP Routing* examines a 1-tuple, *Packet*

Table 1. Application Properties

Application	Data Structure	Packet Header Field	Description
IP Routing	Routing Table	IP Destination Address	Dest. address field is used to lookup next hop info. in the routing table.
Packet Classification	Classification Table	IP Source Address TCP Source Port IP Destination Address TCP Destination Port Protocol Number	The 5-tuple is used to lookup the classification table and identify the flow the packet belongs to.
NAT Protocol (In)	Port Mapping Table	IP Destination Address TCP Destination Port	The destination address and port fields are used to lookup for the internal destination address and port inside the subnet for incoming traffic.
NAT Protocol (Out)	Port Mapping Table	IP Source Address TCP Source Port	The source address and port fields are used to lookup for a translated source port for outgoing traffic.

Table 2. Packet Stream Characteristics

Packet Stream	Source	Num. of Packets	Num. of Distinct Destinations
1	19991129-134258-0	17045569	8920
2	20000112-111915-0	17934563	14033
3	20000117-095016-0	18433128	11746
4	20000126-205741-0	18823943	9012



(a) IP Routing: Destination Field

(b) Packet Classification: 5-Tuple

(c) NAT Protocol (in) : 2-Tuple

(d) NAT Protocol (Out) : 2-Tuple

Fig. 1. Average Percentage of Packets with n-tuple Values from the Top 16/32/64/128 Frequently Observed n-tuple Values in 16K Packet Intervals

Classification examines a 5-tuple, and *NAT Protocol* examines a 4-tuple of fields. To measure this locality we divided the packet stream into intervals of 16K of consecutive packets. We determined the top 16, 32, 64, and 128 values for the above n-tuples for each interval by examining the packets in each interval.

Table 3. Average Unique Reuse Distance

Stream	Top 16	Top 32	Top 64	Top 128
1	6.65	8.23	9.49	10.03
2	12.43	14.57	17.08	19.97
3	5.14	6.09	7.01	7.56
4	6.34	7.28	8.11	8.49

(a) Routing Lookup: 1-tuple

Stream	Top 16	Top 32	Top 64	Top 128
1	9.12	11.69	14.56	17.02
2	16.39	20.72	26.92	31.43
3	8.41	10.36	12.53	14.71
4	9.31	11.07	12.88	14.70

(b) NAT (out) : 2-Tuple

Stream	Top 16	Top 32	Top 64	Top 128
1	17.68	21.11	24.41	27.29
2	10.22	12.71	15.70	18.39
3	12.80	15.64	18.37	20.90
4	10.08	12.03	13.91	15.82

(b) NAT (in) : 2-Tuple

Stream	Top 16	Top 32	Top 64	Top 128
1	20.19	24.02	28.09	31.90
2	20.35	25.88	32.32	36.70
3	15.24	18. 50	21.84	24.69
4	12.57	14.78	16.84	19.20

(c) Packet Classification: 5-Tuple

Figure 1 plots the percentage of packets that match the top 16, 32, 64 and 128 n-tuples over time for the packet stream trace 1 in Table 2. We omit the graphs for the other packet stream traces as they are similar. For this trace, even though there was 8920 unique destination addresses, in the 16K intervals shown, capturing the top 128 n-tuples accounted for 60% to 100% of the packets. This shows that within 16K intervals a significant amount of value reuse occurs for the applications examined. In addition, we also examined varying the interval size from 1K to 1024K packets, and the results were roughly the same. Therefore, the degree of value locality in these packet streams is quite significant.

We now consider the *unique recurrence distance* between a pair of packets with the same n-tuple value. This is calculated as the number of packets between two distinct occurrences of two packets with the same top n-tuple value. This notion is similar to the working set size of a cache and more accurately reflects the value locality. Table 3 shows the average number of packets between unique recurrences for the n-tuple values that are in the top 16, 32, 64 and 128 reoccurring tuples over 16K packet intervals. Results are shown for all four packet stream traces from Table 2. The results show that the most frequently occurring n-tuples, Top 16, have a smaller reuse distance than the Top 128 n-tuples.

The reuse distance and frequency of value locality results tell us that a cache of a small size is enough to catch a reasonable amount of value locality.

5 Computation Reuse Cache

In this section we present our computation reuse cache design for network processors. The cache design we propose has two distinct features. First, the cache can be used across several applications. This goal is achieved by making the cache programmable (i.e. the composition of the tag and data parts can be changed from one application to next). Second, this cache is designed to eliminate redundant computation associated with the network processing data-plane algorithm. Because of the repeated occurrence of the same packet header fields, the data-plane algorithm often performs redundant computation. The computation reuse

cache remembers previously performed computations so that later redundant data-plane algorithm queries can be avoided.

It is important to note that the caching we perform corresponds to a coarse-grained computation made up of many instructions. Our approach is similar in idea to the dynamic instruction reuse techniques by Sodani and Sohi [23]. They focused on identifying arbitrary dependency changes of instructions in high performance processors that are performing redundant calculations. If these can be discovered, then the whole computation can be avoided. In our paper we exploit this concept with a programmable computation reuse cache. Our level of reuse focuses on reusing complete set of function calls (the data-plane algorithm query), instead of arbitrary dependency chains as examined by Sodani and Sohi. Our computation reuse cache is set up specifically for each data-plane algorithm to exploit reuse in its calculations. We focus on using this to streamline the processing of packets in order to save energy.

For redundancy elimination, a cache line is designed to contain the input (tag) and output (data) of the computation. The input is the relevant fields (n-tuple) in packet headers, working as the tags of the cache line, and the cache line data is the computation result. The cache is configurable by the application. Each data-plane algorithm is broken into three stages – preprocessing, data-plane processing, and post-processing, and a packet goes through these three stages when being processed by the data-plane algorithm. In the preprocessing stage of a packet, when portions of the packet are read in, we form a tag from the relevant fields. Before the preprocessing stage ends, a lookup in the cache with the n-tuple for the packet is triggered. If a cache hit occurs, the hardware automatically changes the control flow to the post-processing stage for that packet's processing thread and avoids the whole execution of data-plane processing phase. During this post-processing, the computation result is copied from the cache block to registers. In case of a cache miss, the processing continues normally and when the starting point of post-processing phase is reached, the hardware updates the cache with the computation result. The cache is developed in such a way that the worst case throughput of the processing phase is not increased. This is implemented with software assistance and dedicated hardware.

A special register called the *jump target register* (JTR) is added in the micro-engine controller for each thread (hardware context). JTR remembers the starting point of the postprocessing phase of the algorithm so that in case of a cache hit, the control can be directly transfered to this point. In case of cache miss, at this instruction address the computation and its result are sent to the cache for updating the cache line. This register is set during the initialization part of the data-plane algorithm.

The cache is configurable in both the input and output of the processing phase. When initializing the data-plane algorithm at boot time, the configuration of the cache is specified. Masks in the cache are set up so that appropriate header fields can be extracted from the packet for use as the index into the cache. The starting point of postprocessing stage is put into JTR.

When a packet is preprocessed, the data to perform data-plane algorithm comes from the packet header. This same data is used to form the cache index and tag. Therefore, the memory read instructions in the preprocessing phase are marked so that as they read the relevant packet header fields, those values are also sent to the computation reuse cache. Multiple memory read instructions may need to be marked depending upon the number of fields in the n-tuple which acts as the input to the data-plane algorithm. The cache is setup to receive for each thread the input n-tuple, and the arrival of the last value triggers the computation reuse cache lookup.

When performing the computation reuse cache lookup, the n-tuple is hashed into a cache set index, and then the n-tuple is compared to all of the tags in the set. Note, the tag is itself a n-tuple. If there is a hit for the n-tuple, the result data is copied into registers for the postprocessing phase and control is transfered to the instruction in the JTR register. In case of a cache miss, the cache tag is updated with the n-tuple, and the cache block is updated with the result data that becomes available after the processing phase of the data-plane algorithm.

The results in this paper assume that there is a separate computation reuse cache for each data-plane algorithm examined.

6 Using Fetch Gating

In this section, we describe our approach for performing fetch gating while using the computation reuse cache. Fetch gating is a form of pipeline gating proposed by Manne et. al. [19]. Pipeline gating was proposed to stop fetching and executing instructions down wrong (branch mispredicted) paths of execution in order to save energy. We use the same concept here to stall fetch, resulting in energy savings, for the data-plane algorithm when the calculations can be reused due to the computation reuse cache hits. This is possible since the overall network processor is balanced, and if the data-plane algorithm can reuse and jump ahead in its computations this creates slack in the data-plane algorithm's schedule. It can therefore gate execution while the rest of the network processor continues to process the packets for the overall designed throughput.

Figure 2 gives a simplified high level view of a network processor using the computation reuse cache for a data-plane algorithm. In our design we assume that the data-plane algorithm has two queues connecting it to the other parts of the processor so that it can be scaled independently of other stages. The input queue to the data plane algorithm initiates the fetch gating logic. Each time there is a change in the queue length, the hardware decides whether to perform fetch gating or keep the processor in normal state. The fetch gating algorithm currently uses two levels. One corresponds to the standard operation when no fetch gating is performed, and the other corresponds to the fetch gated power saving mode. During the latter mode, no fetching or execution will be performed by the fetch gated data-plane algorithm micro-engine.

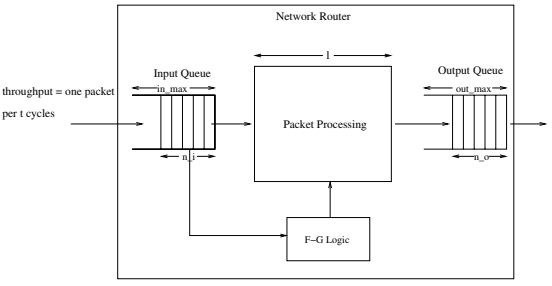


Fig. 2. Network Router with Fetch Gating (F-G) logic

The underlying principle of our approach is to perform fetch gating based upon the occupancy of the input queue shown in Figure 2. In a balanced network processor design, the occupancy of the input queue should be low. If this is the case, and we are getting a reasonable number of computation reuse cache hits, we can save energy by applying fetch gating to the micro-engine. This can continue up to a point. Once the input queue becomes occupied enough, fetch gating is turned off in order to still provide worst-case throughput guarantees and to prevent packets from being dropped.

In Figure 2, we assume that the number of packets in the input queue is n_i. The fetch gating algorithm checks whether the input queue size, n_i, is smaller than an worst case throughput input queue size threshold. If the input queue to the data-plane algorithm micro-engine has enough empty slots to guarantee worst case throughput for its implementation, then the micro-engine is allowed to be in fetch-gated, else the algorithm performs normal fetch.

7 Experimental Evaluation

We use the Nepsim [17] simulator in our experiments. Nepsim is a cycle-accurate simulator of the Intel IXP1200 network processor. We modified Nepsim to represent a balanced network processor with higher throughputs and extended it with our computation reuse cache and our fetch gating algorithm. For our results we model a 30 cycle on-chip SRAM latency to store the data structure for the data-plane algorithm being examined.

In our evaluation, we use four benchmarks which are modified from Intel's Workbench suite or developed by ourselves and migrated to run on the Nepsim simulator. They are IP-Lookup, Packet Classification, and NAT Protocol (in and out). The properties of the application are summarized in Table 4. These applications have the code size of 200 to 300 instructions, shown in the first column. We also show the worst-case packet processing time (latency) that was observed across the four traces in the second column. For IP-Lookup this is 646 cycles and for packet classification it is 1160 cycles. This processing time is considered as the time period between when a packet enters into the processing stage and when it leaves the processing stage (i.e., not counting the time that

Table 4. Application Properties

Applications	Code Size	Proc. Time (Worst)	Proc. Time (Average)	#SRAM Refs
ip-lookup	291	646	435	5
classification	254	1160	1010	13
nat-in	205	754	603	6
nat-out	205	757	597	6

Table 5. Computation Reuse Cache Hit Rate

Applications	Packet Stream Trace			
	1	2	3	4
ip-lookup	68.43%	70.18%	88.89%	85.79%
classification	60.03%	84.97%	77.42%	77.42%
nat-in	74.70%	67.87%	84.45%	82.57%
nat-out	52.52%	82.78%	71.68%	71.68%

Table 6. Program Behavior with Computation Reuse Cache

Applications	ME % Total Energy	Time No Cache	Time Reduction	Cycles Gated	ME Energy Reduction
ip-lookup	33.47%	435	18.62%	22.45%	16.61%
classification	29.66%	1010	47.23%	30.78%	18.89%
nat-in	27.71%	603	51.58%	45.82%	37.69%
nat-out	29.38%	597	41.37%	42.64%	28.43%

the data-plane algorithm is spinning and waiting for the packet to arrive). It also does not include the cycles spent receiving and transmitting the packet. We also show the average-case packet processing time in the third column – the average is computed over the four traces. The last column shows the number of SRAM references needed in the worst case for the algorithm.

7.1 Cache Behavior

Table 5 shows the hit rate when using a 64 entry direct mapped computation reuse cache when the four different packet streams from Table 2 are used as input. The cache hits vary between 52% and 89%. These results are also consistent with the results of our packet value locality study.

7.2 Fetch Gating and Energy Savings

Table 6 shows the effect of cache hits on the data-plane algorithm. These results were obtained by running trace 1 from Table 2 through each of the algorithms. The first column, ME Energy, gives the energy used by the micro-engine on which the data-plane algorithm is being run as a percentage of total energy of the network processor. The second and third columns show the average packet processing latency (cycles) in the absence of cache and the percentage reduction

in this time when the cache is used. The primary benefit of a computation reuse cache hit is the reduction in instructions executed in order to perform the packet processing. For IP-Lookup, we observe close to 19% reduction while for classification we achieve roughly 47% reduction in processing time. The results tell us that the computation reuse cache can provide significant reductions in instructions executed and this creates significant slack.

The last two columns in Table 6 show the results of applying our fetch gating algorithm in the prior section. The energy is recorded using the models provided in the Nepsim [17] simulator augmented to take into consideration the cache and our network processor changes. The fourth column shows the percentage of cycles the algorithm was fetch gated. The last column shows the percentage of energy savings when using the fetch gating when compared to the energy used for just that micro-engine shown in column one. For these results, we only examine one data-plane algorithm at a time, and the computation reuse cache is 64-entry and directed mapped. The results show that for IP-Lookup we spend 22% of the cycles fetch gated for the data-plane algorithm micro-engine and achieve an energy savings of nearly 17%. This results in an overall network processor energy savings of 6%. Across all applications, the computation reuse cache allows 22% to 46% of packet processing to be performed in fetch gated mode which produces 17% to 38% in energy savings for the micro-engines. Note, that slack is generated by reducing the amount of computation needed for a packet when there is a hit in the computation reuse cache, and then energy savings comes from using fetch gating to exploit this slack.

8 Conclusions

High end network processors are built with a balanced processor design, where using a traditional cache for the data-plane algorithm will not increase throughput. These processors are built such that there is sufficient threading (packet parallelism) to hide each data-plane algorithm SRAM lookup latency, so a traditional cache cannot increase throughput.

In this paper we presented a computation reuse cache, where a hit hides the full data-plane algorithm processing of the packet, not just one SRAM lookup as in a traditional cache. This is accomplished by having a cache block contain the input as the tag and output as the data of the data-plane algorithm computation. Therefore a complete query performed by the data-plane algorithm takes one cache access if there is a hit. Slack is therefore generated by reducing the number of instructions executed when there is a hit. This reduction in number of instructions allows us to exploit the slack to save energy through fetch gating for the data-plane algorithm micro-engine while still matching the worst case throughput guarantees of the rest of the processor. Overall, the computation reuse cache allowed 22% to 46% of the execution time to be performed in fetch gated mode with 17% to 38% reduction in data-plane algorithm energy across the different algorithms examined.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper, and early feedback from Nathan Tuck on this topic. This work was funded in part by NSF grant CNS-0509546, and grants from Microsoft and Intel Corporation to the University of California, San Diego and NSF grant CCF-0208756, and grants from Intel Corp., IBM Corp., and Microsoft to the University of Arizona.

References

1. Auckland-II Trace Archive, http://pma.nlanr.net/Traces/long/auck2.html
2. A. Baniasadi and A. Moshovos, "Instruction Flow-based Front-end Throttling for Power-Aware High-Performance Processors," *International Symposium on Low Power Electronics and Design*, August 2001.
3. P. Brink, M. Casterlino, D. Meng, C. Rawal, and H. Tadepalli, "Network Processing Performance Metrics for IA- and IXP-based Systems," *Intel Technology Journal*, Vol. 7, No 4, Nov. 2003.
4. N. Brownlee and M. Murray, "Streams, Flows and Torrents," *Passive and Active Measurement Workshop*, April 2001.
5. A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi, "An Adaptive Issue Queue for Reduced Power at High Performance," *International Workshop on Power-Aware Computer Systems*, November 2000.
6. T. Chiueh and P. Pradhan, "High Performance IP Routing Table Lookup Using CPU Caching," *IEEE Conference on Computer Communications*, April 1999.
7. K. Claffy, "Internet Traffic Characterization," Ph.D. thesis, Univ. of California, San Diego, 1994.
8. M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," *ACM Conference of the Special Interest Group on Data Communication*, September 1997.
9. Y. Ding and Z. Li, "A Compiler Scheme for Reusing Intermediate Computation Results," *International Symposium on Code Generation and Optimization*, March 2004.
10. W. Doeringer, G. Karjoth, and M. Nassehi, "Routing on Longest Matching Prefixes," *IEEE/ACM Transactions on Networking*, Vol. 4, No. 1, pages 86-97, Feb. 1996.
11. K. Egevang and P. Francis, "The IP Network Address Translator (NAT)," RFC 1631, May 1994.
12. P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *ACM Conference of the Special Interest Group on Data Communication*, September 1999.
13. P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network*, Vol. 15, No. 2, pages 24-32, Sept. 2001.
14. E. Johnson and A. Kunze, *IXP2400/2800 Programming*, Intel Press, 2003.
15. T. Karkhanis, J.E. Smith, and P. Bose, "Saving Energy with Just In Time Instruction Delivery," *International Symposium on Low Power Electronics and Design*, August 2002.
16. K. Li, F. Chang, D. Berger, and W. Feng, "Architectures for Packet Classification Caching," *The 11th IEEE International Conference on Networks*, Sept./Oct. 2003.

17. Y. Luo, J. Yang, L. Bhuyan, and L. Zhao, "NePSim: A Network Processor Simulator with Power Evaluation Framework," *IEEE Micro Special Issue on Network Processors for Future High-End Systems and Applications*, Sept./Oct. 2004.
18. Y. Luo, J. Yu, J. Yang, and L. Bhuyan, "Low Power Network Processor Design Using Clock Gating," *42nd Annual Design Automation Conference*, June 2005.
19. S. Manne, A. Klauser, and D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction," *International Symposium on Computer Architecture*, June 1998.
20. G. Memik and W.H. Mangione-Smith, "Improving Power Efficiency of Multi-Core Network Processors Through Data Filtering," *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Oct. 2002.
21. C. Partridge, "Locality and Route Caches", *NSF Workshop on Internet Statistics Measurement and Analysis*, Feb. 1996. (http://www.caida.org/ISMA/Positions/partridge.html).
22. T. Sherwood, G. Varghese, and B. Calder, "A Pipelined Memory Architecture for High Throughput Network Processors," *International Symposium on Computer Architecture*, June 2003.
23. A. Sodani and G.S. Sohi, "Dynamic Instruction Reuse," *International Symposium on Computer Architecture*, pages 194-205, June 1997.
24. W. Szpankowski, "Patricia Tries Again Revisited", *Journal of the ACM*, Vol. 37, No. 4, pages 691-711, October 1990.

Dynamic Evolution of Congestion Trees: Analysis and Impact on Switch Architecture*

P.J. García[1], J. Flich[2], J. Duato[2], I. Johnson[3], F.J. Quiles[1], and F. Naven[3]

[1] Dept. de Informática. Univ. Castilla-La Mancha 02071-Albacete, Spain
{pgarcia, paco}@info-ab.uclm.es
[2] Dept. of Computer Science, Univ. Politécnica de Valencia 46071-Valencia, Spain
{jflich, jduato}@disca.upv.es
[3] Xyratex, Haven, United Kingdom
{Ian_Johnson, Finbar_Naven}@xyratex.com

Abstract. Designers of large parallel computers and clusters are becoming increasingly concerned with the cost and power consumption of the interconnection network. A simple way to reduce them consists of reducing the number of network components and increasing their utilization. However, doing so without a suitable congestion management mechanism may lead to dramatic throughput degradation when the network enters saturation. Congestion management strategies for lossy networks (computer networks) are well known, but relatively little effort has been devoted to congestion management in lossless networks (parallel computers, clusters, and on-chip networks). Additionally, congestion is much more difficult to solve in this context due to the formation of congestion trees.

In this paper we study the dynamic evolution of congestion trees. We show that, contrary to the common belief, trees do not only grow from the root toward the leaves. There exist cases where trees grow from the leaves to the root, cases where several congestion trees grow independently and later merge, and even cases where some congestion trees completely overlap while being independent. This complex evolution and its implications on switch architecture are analyzed, proposing enhancements to a recently proposed congestion management mechanism and showing the impact on performance of different design decisions.

1 Introduction

Designers of large parallel computers, clusters, and on-chip networks are becoming increasingly concerned with the cost and power consumption of the interconnection network. Effectively, current interconnect technologies (Myrinet 2000 [24], Quadrics [28], InfiniBand [16], etc.) are expensive compared to processors. Also, power consumption is becoming increasingly important. As link speed increases, interconnects consume a greater fraction of the total system power [30]. Moreover, power consumption in current high-speed links is almost independent of link utilization. In order to reduce system power consumption, researchers have proposed using frequency/voltage scaling techniques [30]. Unfortunately, these techniques are quite inefficient due to their

* This work was supported by CICYT under Grant TIC2003-08154-C06, by UPV under Grant 20040937 and by Junta de Comunidades de Castilla-La Mancha under Grant PBC-05-005.

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 266–285, 2005.

slow response in the presence of traffic variations and the suboptimal frequency/voltage settings during transitions [34].

As the interconnection network has traditionally been overdimensioned, a simple way to reduce cost and power consumption is reducing the number of network components (i.e. switches and links) and increasing their utilization. However, this solution will increase network contention that, as traffic is usually bursty, may lead to congestion situations. In general, congestion will quickly spread through the network due to flow control, forming congestion trees. The main negative effect of these situations happens when packets blocked due to congestion prevent the advance of other packets stored in the same queue, even if they are not going to cross the congested area. This effect is referred to as head-of-line (HOL) blocking, and may produce a dramatic network throughput degradation.

The behavior of network congestion depends on switch architecture. HOL blocking is one of the main problems arising in networks based on switches with queues at their input ports (Input Queuing, IQ switches), because blocked packets destined to congested output switch ports prevent the advance of packets destined to other non-congested output ports. This problem can limit switch throughput to about 58% of its peak value [17]. So, from the point of view of switch architecture, HOL blocking will affect greatly to IQ switches and also to switches with queues at their input and output ports (Combined Input and Output Queuing, CIOQ switches). These architectures are different from that of traditional switches in communication networks, that uses queues only at their output ports (Output Queuing, OQ switches). The OQ scheme has become infeasible because requires the switch to operate at a much faster speed than the links in order to handle all the possible packets concurrently arriving at the input ports[1], and link speed in current high-speed interconnects is on the order of Gbps.

So, as the switch architecture used in most recent designs follows the IQ or CIOQ model, HOL blocking could be a serious problem in modern interconnects. In CIOQ switches, HOL blocking can be reduced by increasing switch speedup. A switch with a speedup of S can remove S packets from each input and deliver up to S packets to each output within a time slot, where a time slot is the time between packet arrivals at input ports. Note that OQ switches have a speedup of N while IQ switches have a speedup of 1. CIOQ switches have values of S between 1 and N. However, for larger values of S, switches become too expensive, and practical values are between 1.5 and 2. Therefore, the switch speedup increase is limited, and HOL blocking should be controlled by a suitable congestion management mechanism.

It should be noted that although congestion management strategies for computer networks are well known, congestion dynamics in networks (whether on-chip or off-chip) for parallel computers is completely different from the one in computer networks because packets are not dropped when congestion arises. Moreover, congestion is more difficult to be solved due to the quick growth of congestion trees. Additionally, as congestion has not been a problem until recently (networks were overdimensioned), there are relatively few studies on congestion trees and their dynamics [27]. In fact, despite the different queue architectures, it is common belief that congestion trees always grow from the root to the leaves. This is true in networks with OQ switches, but this is not

[1] Theoretically, a $N \times N$ OQ switch must work N times faster than the link speed.

the case for networks with CIOQ switches. In this case, as HOL blocking may occur throughout the network, congestion trees may evolve in several ways.

Thus, we believe that an in-depth study of congestion trees and their dynamic evolution will help in finding better solutions for congestion management in interconnection networks with CIOQ switches. In this paper we take on such challenge. We show that congestion trees do not only grow from the root toward the leaves. There exist relatively frequent conditions under which trees grow from the leaves to the root, cases where several congestion trees grow independently and later merge, and even cases where a congestion tree grows in such a way that completely overlaps with a subset of another congestion tree while being independent from it. This complex evolution has some implications on the way congestion management strategies should be designed. Moreover, some design decisions (e.g., where congestion detection should be performed) also depend on switch architecture, as a consequence of the way congestion trees evolve.

So, in addition to study the evolution of congestion trees, we present some enhancements to a recently proposed congestion management mechanism [12]. These enhancements will provide support for efficiently handling the situations described in the analysis of congestion dynamics.

To sum up, the main contributions of this paper are: 1) a detailed analysis of the dynamics of congestion trees when using CIOQ switches, showing their complex evolution, 2) the proposal of several enhancements to a previously proposed congestion management mechanism so that it will efficiently handle the complex situations (we also show the relationship between those enhancements and switch architecture), and 3) an evaluation of the impact on performance of the enhancements.

The rest of the paper is organized as follows. In Section 2, related work is presented. In Section 3, the dynamic evolution of congestion trees is analyzed in depth. Then, in Section 4, the proposed enhancements, based on the analysis of the dynamic evolution of congestion trees, are presented. Evaluation results are presented in Section 5. Finally, in Section 6, some conclusions are drawn.

2 Related Work

The formation of congestion (or saturation) trees on multistage interconnection networks (MINs) under certain traffic conditions has deserved the attention of researchers for many years [27]. A large number of strategies have been proposed for controlling the formation of congestion trees and for eliminating or reducing their negative effects. Many of them consider congestion in multiprocessor systems, where saturation trees appear due to concurrent requests to the same shared memory module. In [9], a taxonomy of hot-spot management strategies is proposed, dividing them into three categories: avoidance-based, prevention-based and detection-based strategies. Although different taxonomies are possible for other environments [40], the former classification is roughly valid also for non-multiprocessor-oriented congestion management techniques.

Avoidance-based strategies require a previous planning in order to guarantee that congestion trees will not appear. Some of these techniques are software-based [41,5], while others are hardware-oriented [39]. In general, these strategies are related to quality of service requirements.

Prevention-based strategies control the traffic in such a way that congestion trees should not happen. In general, decisions are made "on the fly", based on limiting or modifying routes or memory accesses. These techniques can be software-based [15] or hardware-oriented [1,29].

When detection-based strategies are used, congestion trees may form, but they can be detected in order to activate some control mechanism that should solve the problem. Usually, this kind of mechanism requires some feedback information. For instance, it is possible to measure the switch buffer occupancy [38,22] or the amount of memory access requests [29] in order to detect congestion. Later, a notification is sent to the sources injecting traffic or to the processors requesting memory accesses, in order to cease or reduce their activity. Notifications could be sent to all the sources [36] or just to those that cause the congestion [19]. Other mechanisms [8,23,3,4] notify congestion just to the endpoints attached to the switch where congestion is detected.

Recently, a new mechanism have been proposed for networks with CIOQ switches [12]. It is based on dynamically separating the traffic belonging to different congestion trees, eliminating so the HOL blocking introduced by congestion. Additionally, many proposals minimize or eliminate HOL blocking regardless of its cause. Some of them focus on HOL blocking formed at the switch level [2,35,32,7,21] while others at the entire network [6,18]. The use of non-blocking topologies [11] also eliminates HOL blocking. Finally, other strategies like fully adaptive routing [10,20,14,37,31] or load balancing techniques [13,31] may help to delay the appearance of congestion.

It is not in the scope of the present paper to discuss the advantages and drawbacks of all these strategies. However, we must remark that, as far as we know, none of them take into account the formation process of congestion trees. This paper is focused on analyzing this important subject.

3 Dynamic Evolution of Congestion Trees

In this section we will analyze the different scenarios that may arise in the formation of congestion trees for networks with CIOQ switches. In particular, we will focus on two key aspects that greatly influence how congestion trees are formed. The first one is the architecture of the switch whereas the second one is the traffic pattern. As we will show later, the scenarios analyzed in this section must be properly handled by a congestion control mechanism in order to be effective.

3.1 Traditional View

Traditionally, it has been thought that congestion trees form as follows: the root of the congestion tree (e.g., an output port at some switch) becomes congested and, due to the use of flow control, congestion spreads from the root to the leaves. However, this situation happens only in a particular scenario that rarely occurs: when the different traffic flows that form the tree join only at the root. Figure 1.a shows an example: five flows form a congestion tree by meeting at the root switch. The sum of the injection rate of all the flows is higher than the link bandwidth, thus a congestion tree is formed. Before congestion occurs, all the queues used along the path followed by the flows are

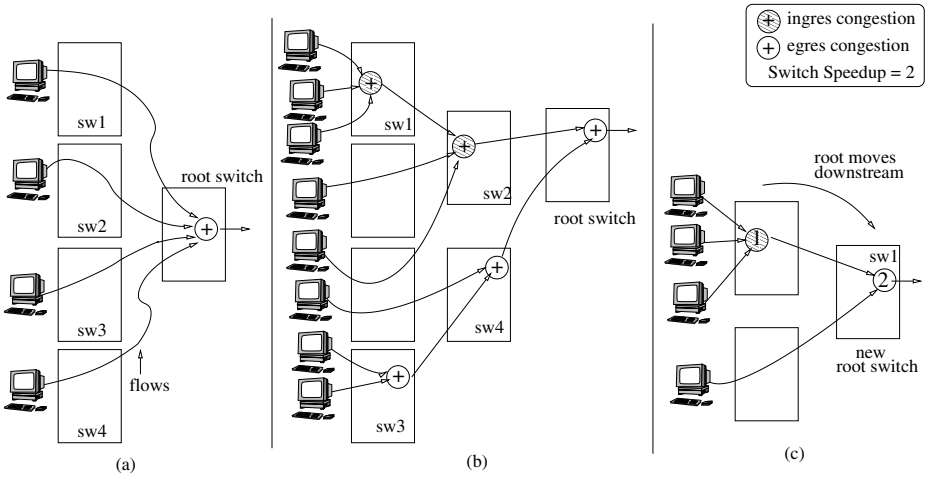


Fig. 1. Different dynamic formations of trees. Speedup is two. (a) Traditional view about the formation of a congestion tree, (b) Different locations (ingress and egress) where congestion is detected for one tree, and (c) the root switch moves downstream.

nearly empty because there is enough bandwidth. However, when flows meet at the root switch, the rate at which packets arrive is higher than the rate at which packets can be transmitted through the output port. Thus, queues will start to fill at the root switch (assuming an appropriate crossbar speedup, egress queues fill first, and later ingress queues), and thus, congestion will begin. This situation will last until the queues at the root switch fill up and, at this time, queues at the previous switches of the root switch will start to queue packets (again, first at the egress queues and then at the ingress queues). This is because the available bandwidth at the output port of the root switch will be divided among the different flows and will be lower than the bandwidth required by each flow. At the end, the congestion tree will grow from the root to the leaves.

Traditionally (and in the former example), it has also been assumed that in a CIOQ switch with speedup, packets will start queuing at the egress ports when congestion forms. However, as will be seen in the next section, this may not always be true.

3.2 Effect of Switch Architecture on the Dynamics

As mentioned above, the switch architecture, specifically the speedup, can alleviate the possible HOL blocking at the input ports. However, it is not completely eliminated. In particular, depending on the number of flows and the rate at which they arrive at a switch, the formation of a congestion tree will be different. Figure 2.a shows an example where two flows, injected at the full rate and headed to the same destination join at a switch with no speedup. As the total reception is higher than the rate at which packets are forwarded to the egress side, packets are queued at the ingress side. Thus, congestion will occur at the ingress side of the switch. However, if speedup is used (Figure 2.b, speedup of 2) congestion occurs at the egress side. As the number of flows arriving at

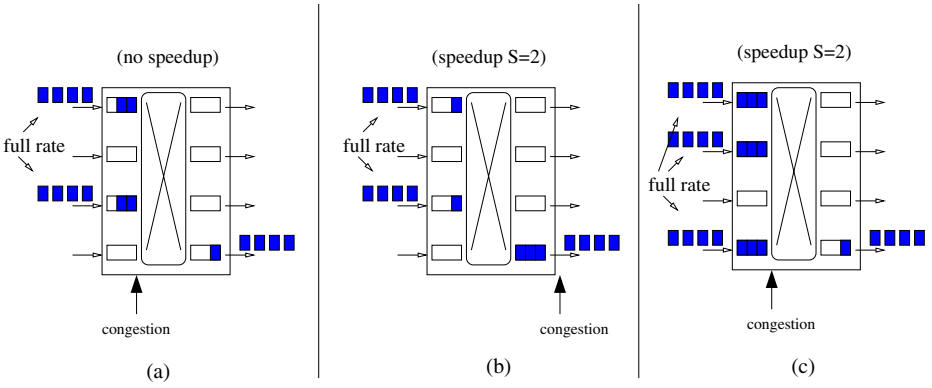


Fig. 2. HOL blocking within a switch with different speedups and different flows

the root switch is equal to or less than the speedup, the switch can forward the flows to the egress side at the same rate they are arriving. But, as the output port bandwidth is half the internal switch bandwidth, packets start to queue at the egress side.

It should be noted that this situation depends on the number of incoming flows, the rate at which they arrive, and the switch speedup. Although the speedup can be increased, it drastically increases the switch cost. So, limited speedups are often used (i.e. not higher than 2). As an example, Figure 2.c shows a case where three flows headed to the same destination arrive at a switch with speedup of 2. The rate at which packets arrive is higher than the rate at which they are forwarded to the egress side. Thus, congestion arises at the ingress side, contrary to the common belief.

This effect may also occur at several points along a congestion tree. In Figure 1.b we can observe a congestion tree formed by eight flows heading to the same destination. Switch speedup is 2. Three flows merge at $sw1$, and other flows merge at different points, finally meeting all together at the root switch. In this situation, at $sw1$ and $sw2$, congestion first occurs at the ingress side. Once all the flows arrive at the root switch they merge and congestion occurs again. However, in this case, congestion first occurs at the egress side, and so happens at $sw3$ and $sw4$.

Therefore, the speedup affects the way congestion trees form. In particular, congestion may first occur at ingress or egress side and, at the same time, different local congestion spots may arise during the formation of the congestion tree. Although the different local congestion spots can be initially viewed as independent congestion trees, they end up being part of a single and larger congestion tree. Whether only the complete congestion tree or each local congestion spot should be treated by a congestion control mechanism will be discussed in section 4. This decision will significantly affect the effectiveness of the congestion control mechanism.

3.3 Impact of Traffic Patterns on the Dynamics

In the previous scenario it was assumed that all the flows were injecting packets at the same rate and started at the same time. This could be the case when executing a barrier

synchronization among different processes in a multiprocessor system. However, other scenarios may arise where different flows contributing to the same congestion tree start at different times and inject at different rates. This will lead to more sophisticated dynamics in the formation of congestion trees. As an example, consider Figure 1.c where a first congestion tree made up of three flows is created. These flows, headed to the same destination, join together at the first stage of the network. Thus, the root switch is at the first stage (as speedup of 2 is assumed, congestion first occurs at the ingress side). Later, an additional flow headed to the same destination is injected. However, contrary to the other flows, it merges with them at the second stage. Assuming that all the flows are injecting at the full injection rate, the bandwidth at the egress port of $sw1$ must be shared among the four flows. Thus, a new congestion is formed at the egress side of $sw1$. In fact, two congestion trees have been created at different instants, but the first one later becomes a subtree of a larger tree. This situation can be viewed as if the root of the congestion tree had moved downstream in the network.

Another scenario occurs when two congestion trees overlap. This may happen quite frequently in server systems with different disks placed close to each other. As each accessed disk may produce a congestion tree, several trees may overlap.

When two congestion trees overlap, new dynamic behaviors appear. Figure 3.a shows two congestion trees. The one plotted in solid lines ($ct1$) is formed first, whereas the one plotted in dashed lines ($ct2$) appears later. In this situation, $ct1$ has its root switch at $sw1$ and $ct2$ at $sw2$. However, when $ct2$ appears, a new congestion point is located at $sw3$. This new congestion point can be viewed as a new root for both trees, that will finally merge into one. However, as the congestion point has been formed by

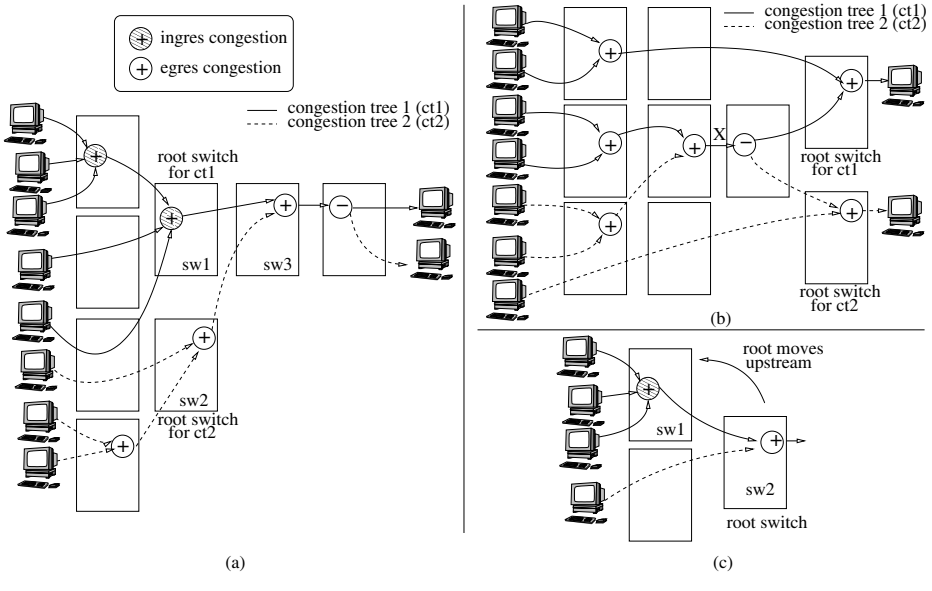


Fig. 3. Different dynamic formations of trees. Speedup is two. (a) two trees overlap and merge, (b) two trees overlap but do not merge, and (c) the root switch moves upstream.

flows headed to different destinations, it could be considered as two different overlapping congestion trees. Therefore, if congestion control mechanisms allocate resources depending on packet destination, separate resources will be needed for each congestion tree. On the other hand, if congestion within the network is going to be treated, this case should be considered as two trees merging into one, thus saving some resources.

Another interesting situation occurs when two congestion trees overlap but do not merge. For instance, once the congestion tree plotted in solid lines ($ct1$) in figure 3.b is formed, a second one ($ct2$, plotted in dots) forms. A $ct2$ branch shares a set of network resources with $ct1$. Thus, point X can be viewed as belonging to both trees. In this situation, it could happen that a congestion control mechanism would consider traffic addressed to $ct2$ passing through X as traffic belonging to $ct1$. As we will show later, a correct differentiation of both trees will improve performance.

Finally, another case occurs when one or several branches of an already formed tree disappear (sources injecting to the congested destination no longer send packets). This case is shown in Figure 3.c. The flow plotted in dashed line disappears and the root switch ($sw2$) no longer experiences congestion. This situation can be viewed as if the root moves upstream the network to $sw1$. However, it should be noticed that while the three flows arriving at $sw1$ keep injecting at the full rate, the queues used at $sw2$ will remain full although it receives packets only from one input port.

To sum up, the root of a congestion tree may move downstream (by the addition of new flows) or upstream (by the collapse of some branches). At the same time, congestion trees may overlap at several network points without merging. And finally, a congestion tree may be formed from local and transient congestion trees that will later merge (due to the limited speedup). Therefore, for all of these cases some kind of architectural support is needed in order to separate each congestion tree and to follow the complex dynamics they may exhibit. As we will see in the evaluation section, keeping track of their dynamics will ensure decisive benefits in terms of network performance.

4 Implications on the Design of Congestion Control Techniques

One of the main objectives of the paper is to develop new mechanisms *able* to keep track of the complex dynamics of congestion trees in networks with CIOQ switches. For this purpose, we present two enhancements to a previously proposed congestion control mechanism referred to as RECN [12]. RECN focuses on eliminating the HOL blocking induced by congestion trees. As we will see in the evaluation, the two new enhancements will be key to achieve maximum performance and will allow RECN to completely eliminate the HOL blocking induced by congestion trees. For the sake of completeness we will first briefly describe RECN and later the two new enhancements.

4.1 RECN (Regional Explicit Congestion Notification)

RECN is based on the assumption that packets from non-congested flows can be mixed in the same queue without significant interference among them. Therefore, RECN focuses on eliminating the HOL blocking introduced by congestion trees. This is accomplished by detecting congestion and dynamically allocating separate buffers for each

congestion tree. By completely eliminating HOL blocking, maximum performance is achieved even in the presence of congestion trees.

RECN has been designed for PCI Express Advanced Switching[2] (AS) [25,26]. Although it could work under different technologies, RECN benefits from the routing mechanisms found in AS. In particular, AS uses source deterministic routing. The AS header includes a turn pool made up of 31 bits that contains all the turns (offset from the incoming port to the outgoing port) for every switch along the path. An advantage of this routing method is that it allows to address a particular network point from any other point in the network. Thus, a switch, by inspecting the appropriate turnpool bits of a packet, can know in advance if it will pass through a particular network point.

RECN adds, at every input and output port of a switch, a set of additional queues referred to as Set Aside Queues (SAQs). SAQs are dynamically allocated and used to store packets passing through a congested point (root of a congestion tree). To do this, a CAM memory is associated to each set of SAQs. A CAM line contains the control info required to identify a congested point and to manage the corresponding SAQ. Additionally, one queue (referred to as normal queue) is used to store non congested packets.

RECN detects congestion only at switch egress ports. When a normal egress queue receives a packet and fills over a given threshold, a notification is sent to the sender ingress port indicating that the output port is congested. This notification includes the routing information (a turnpool and the corresponding mask bits) to reach the congested output port from the notified ingress port (only a turn). Upon reception of a notification, each ingress port allocates a new SAQ and fills the corresponding CAM line with the received turnpool and mask bits. From that moment, every incoming packet that will pass through the congested point (easily detected from the packet turnpool) will be mapped to the newly allocated SAQ, thus eliminating the HOL blocking it may cause. If an ingress SAQ becomes subsequently congested, a new notification will be sent upstream to some egress port that will react in the same way, allocating an egress SAQ, and so on. As the notifications go upstream, the information indicating the route to the congested point is updated accordingly, in such a way that growing sequences of turns (turnpools) and mask bits are stored in the CAM lines. So, the congestion detection is propagated through all the branches of the tree.

To guarantee in order delivery, whenever a new SAQ is allocated, forwarding packets from that queue is disabled until the last packet of the normal queue (at the moment of the SAQ allocation) is forwarded. This is implemented by a simple pointer associated to the last packet in the normal queue and pointing to the blocked SAQ.

RECN implements for each individual SAQ a special Xon/Xoff flow control, that follows the Stop & Go model. This mechanism is different from the credit-based flow control used for normal queues, that considers all the unused space of the port data memory available for each individual queue. If these "global" credits scheme would be used for SAQs, a congested flow could fill the whole port memory very fast, whereas the Xon/Xoff scheme guarantees that the number of packets in a SAQ will be always below a certain threshold.

[2] AS is an open standard for fabric-interconnection technologies developed by the ASI Special Interest Group. It is based on PCI Express technology, extending it to include other features. ASI is supported by many leader enterprises.

RECN keeps track (with a control bit on each CAM line) of the network points that are leaves of a congestion tree. Whenever a SAQ with the leaf bit set to one empties, the queue is deallocated and a notification is sent downstream, repeating the process until the root of the congestion tree is reached. For a detailed description of the RECN mechanism, please refer to [12].

4.2 Proposed Enhancements

In this section two enhancements to RECN are proposed. They will allow RECN to keep track of the dynamics of congestion trees, and thus, to exhibit significantly better performance.

The first enhancement is allowing RECN to detect congestion at switch ingress ports. RECN defines SAQs at ingress and egress ports, but it only detects congestion at egress ports (it is based on the belief that congestion first occurs at egress side). Thus, ingress ports SAQs are allocated only when receiving notifications. We have previously shown that congestion may first occur at ingress ports (for instance, in switches without speedup). In these cases, RECN never detects congestion at the root. Instead, it detects congestion at the immediate upstream switches, but only when the root ingress queues are full, preventing the packet injection from those switches. So, RECN will not react quickly to eliminate HOL blocking at an important part of the tree.

In order to detect congestion at the ingress side, a different detection mechanism must be used. When detecting congestion at the egress side, the congestion point is the output port by itself. However, when an ingress normal queue fills over a threshold, it is because packets requesting a certain output port are being blocked. As packets in the ingress queue can head to different output ports, it is not trivial to decide which one is the congested output port. In response to this requirement, we propose to replace the normal queue at each ingress port by a set of small buffers (referred to as detection queues). So, at ingress ports, the memory is now shared by detection queues and SAQs. The detection queues are structured at the switch level: there are as many detection queues as output ports in the switch, and packets heading to a particular output port are directed to the associated detection queue[3]. By doing this, when a detection queue fills over a given threshold, congestion is detected, and the output port causing the congestion is easily computed as the port associated with that detection queue[4]. Once congestion is detected at an ingress port, a new SAQ is allocated at this port, and the turnpool identifying the output port causing congestion is stored in the CAM line. The detection queue where congestion is detected and the SAQ allocated are swapped. As the new SAQ can now be considered to be congested, a notification is sent upstream. Figure 4 shows the proposed mechanism.

The second enhancement is related to the actions taken upon reception of notifications. RECN does not allocate SAQs for all the notifications received. This is done in order to ensure that no out of order packet delivery is introduced. Figure 5.a shows an example where RECN does not allocate a SAQ when receiving a notification. First, $sw1$

[3] Note that, if there were no SAQs, the memory at ingress ports would follow a Virtual Output Queuing scheme at switch level. Of course, SAQs make a difference.

[4] There are other ways of detecting, at ingress sides, the flows contributing to congestion. Detection queues minimize congestion detection latency, but their use is not mandatory.

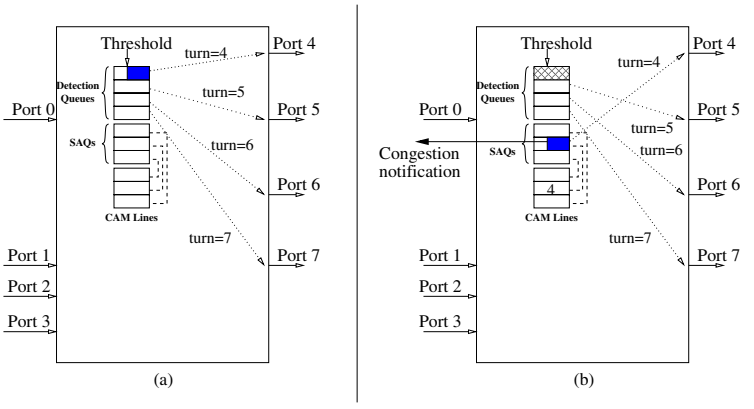


Fig. 4. Mechanism for detecting and handling congestion at the ingress side: (a) Queue status at the detection moment (b) Queue status after detection

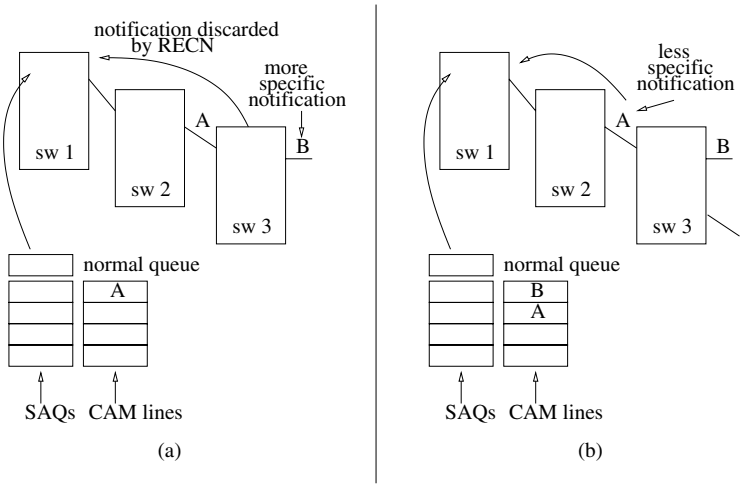


Fig. 5. Basic RECN treatment for (a) more specific and (b) less specific notifications

is notified that point A is congested. Then, it allocates a new SAQ for that congested point and packets going through A will be stored from that moment in that SAQ. Later, point B becomes congested at $sw3$ and notifications are sent upstream, reaching $sw1$. This notification is referred to as being a more specific notification (as B is further away than A). Notice that when the notification arrives to $sw1$ it may happen that packets going through B are stored in the SAQ associated to A (as packets have to pass through A before reaching B). If $sw1$ allocates a new SAQ for B, it may happen that packets later stored in that SAQ could leave the switch before the packets stored at the SAQ associated to A (out of order delivery).

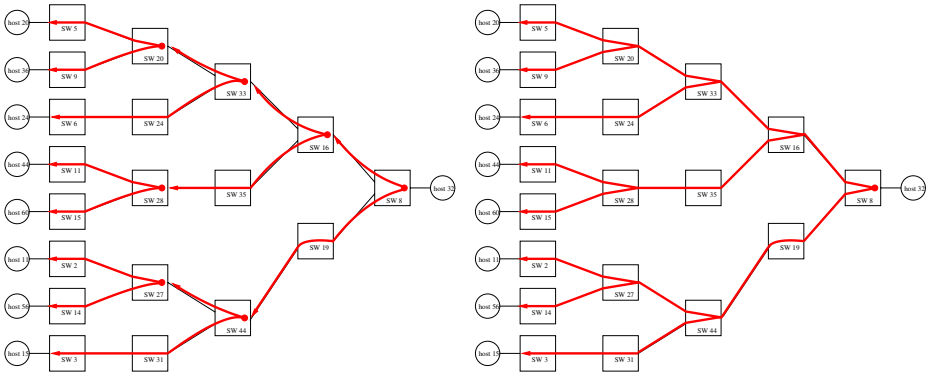
Notice that this fact can lead to RECN to introduce HOL blocking. Indeed, flows not belonging to the first tree (traffic adressed to point B) passing trough point A will

be mapped to the same SAQ (the one associated to A) that flows belonging to the congestion tree. Thus, RECN does not support the downstream tree movement. Also the situation where a congestion tree forms from leaves to root will not be correctly treated.

Figure 5.b shows another situation. In this case $sw1$, having a SAQ allocated for point B, receives a less specific notification (being point A congested). This can be related to the formation of an overlapping congestion tree. In this case, RECN accepts the notification and allocates a new SAQ for point A. In this situation, an arriving packet will be stored in the SAQ whose associated turnpool matches in more turns the packet turnpool. Thus, incoming packets passing trough A and B will be mapped to the SAQ allocated for B, and packets passing through A but not through B will be mapped to the SAQ allocated for A. Notice that in this situation no out of order delivery may be introduced as already stored packets passing through A but not trough B are not mapped to the SAQ asociated to B (they were stored on the normal queue).

Thus, the proposal is to accept all the notifications regardless whether they are more or less specific. In order to deal with out of order issues, when a new SAQ is allocated due to a more specific notification, it must be blocked (must not send packets) until all the packets stored in the SAQ associated to the less specific notification (when the new SAQ is allocated) leave the queue. This can be accomplished by placing in the "old" SAQ a pointer to the new allocated SAQ .

In order to foresee the potential of the proposed enhancements, Figures 6.a and 6.b show how the original (or "basic") RECN and the enhanced RECN mechanisms detect congestion trees. These figures reflect simulation results when a congestion tree is formed by eight sources injecting packets to the same destination (hot-spot) at the full rate of the link. All the sources start sending packets at the same time. Switch speedup has been set to 1.5. In both figures, dots indicate the points (ingress or egress) considered by the mechanism as congestion roots after the tree is formed. The thick arrows indicate the paths followed by congestion notifications (RECN messages) from congestion points, thus indicating where SAQs are allocated for a particular congestion point.



(a) Basic RECN (b) Enhanced RECN

Fig. 6. Congestion tree detection with different RECN versions

As can be observed, with the basic RECN mechanism, one "real" congestion tree is viewed as several subtrees. In particular, seven subtrees are formed, all of them with their roots at egress sides of switches. However, the enhanced RECN mechanism correctly identifies the congestion tree. At the end, only one tree is detected and the root is located at the egress side of switch 8, matching so the real congestion tree.

Although both schemes end up detecting the same congestion (through one tree or several ones), detecting the correct one has powerful benefits. In particular, for the basic RECN mechanism, HOL blocking is not correctly eliminated. As an example, imagine traffic not belonging to the congestion tree that arrives at switch 5 and is passing through some of the detected intermediate congestion points. Those packets will be mapped to the same SAQ used to hold packets addressed to the congested destination, thus introducing massive HOL blocking. As the congestion tree grows in the number of stages, the number of intermediate detected congestion points will increase and, then, more HOL blocking will be introduced. On the other hand, the enhanced mechanism will use a SAQ in all the switches (on every attached port) to store packets exactly destined to the congestion root. Thus the rest of flows will be mapped to the detection queues, and so HOL blocking at the congestion tree will be completely eliminated.

It has to be noted that enabling congestion detection at switch ingress sides also has benefits in the mechanism response. In the basic RECN, intermediate output ports (detected as congested) start to congest only when the ingress queue at the downstream switch totally fills up with congested packets. Thus, at the time congestion is detected some ingress side queues are totally full.

5 Performance Evaluation

In this section we will evaluate the impact of the proposed enhancements over RECN on the overall network performance in different scenarios of traffic load and switch architecture. For this purpose we have developed a detailed event-driven simulator that allows us to model the network at the register transfer level. Firstly, we will describe the main simulation parameters and the modeling considerations we have used in all the evaluations. Secondly, we will analyze the evaluation results.

5.1 Simulation Model

The simulator models a MIN with switches, end nodes, and links. To evaluate the different congestion manegement techniques, we have used several bidirectional MINs (BMINs) shown in Table 1. 8-port switches are used and the interconnection pattern is the perfect shuffle pattern.

In all the experiments deterministic routing has been used. Memories of 32KB have been modeled for both input and output ports of every switch. At each port, the memory is shared by all the queues (normal or detection queues and SAQs) defined at this port at a given time, in such a way that memory cells are dynamically allocated (or deallocated) for any queue when necessary. For the basic RECN mechanism only one normal queue and a maximum of eight SAQs are defined at ingress and egress ports. However, for the enhanced RECN (RECN with the two proposed enhancements) the normal queue at ingress ports has been divided in eight detection queues.

Table 1. Traffic corner cases evaluated

		normal traffic			congestion tree					
Traffic case	network	# srcs	dest	injection rate	# srcs	dest	injection rate	start time	end time	congestion type
#1	64×64	75%	rand	50%	25%	32	100%	variable	variable	incremental
#2	64×64	75%	rand	100%	25%	32	100%	variable	variable	incremental
#3	64×64	75%	rand	50%	25%	32	100%	800 μs	1100 μs	sudden
#4	64×64	75%	rand	100%	25%	32	100%	800 μs	1100 μs	sudden
#5	512×512	75%	rand	100%	6.25%	32	100%	800 μs	1100 μs	sudden
					6.25%	201	100%	800 μs	1100 μs	sudden
					6.25%	428	100%	800 μs	1100 μs	sudden
					6.25%	500	100%	800 μs	1100 μs	sudden
#6	2048×2048	75%	rand	100%	6.25%	32	100%	800 μs	1100 μs	sudden
					6.25%	515	100%	800 μs	1100 μs	sudden
					6.25%	1540	100%	800 μs	1100 μs	sudden
					6.25%	2000	100%	800 μs	1100 μs	sudden

At switches, packets cross from any input queue to any output one through a multiplexed crossbar modeled with two options: no speedup (link and crossbar bandwidth is 8Gbps), or 1.5 speedup (link bandwidth is 8Gbps, crossbar bandwidth is 12Gbps).

End nodes are connected to switches using Input Adapters (IAs). Every IA is modeled with a fixed number of N message admitance queues (where N is the total number of end nodes), and a variable number of injection queues, that follow a scheme similar to that of the output ports of a switch. When a message is generated, it is stored in the admitance queue assigned to its destination, and is packetized before being transferred to an injection queue. We have used 64-byte packets.

We have modeled in detail the two versions of RECN: 1) Basic RECN: detection only at egress side and more restrictive notifications discarded, and 2) Enhanced RECN: detection at ingress ports enabled and more restrictive notifications accepted, preserving in-order delivery. For comparison purposes we also evaluate the Virtual Output Queuing (VOQ) mechanism at the switch level [2]. This method will be referred to as VOQsw.

5.2 Traffic Load

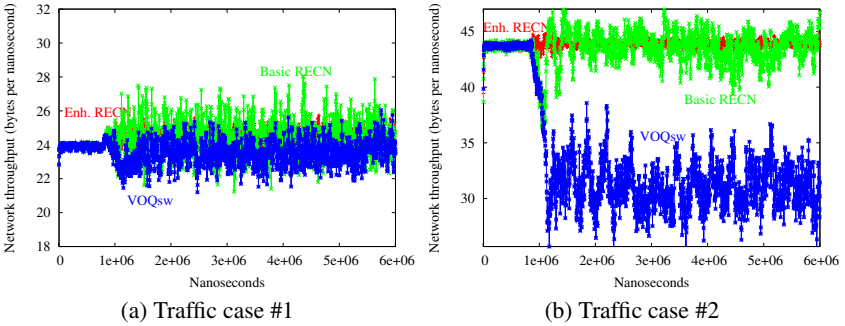
In order to evaluate the RECN mechanisms, two different scenarios will be analyzed. First, well-defined synthetic traffic patterns will be used. Table 1 shows the traffic parameters of each traffic case. For each case, there will be 75% of sources injecting traffic to random destinations during all the simulation period. These nodes will inject traffic at different rates of the link depending on the traffic case. In all the cases, the rest of sources (25%) will inject traffic at the full rate to the same destination (medium networks, traffic cases #1 to #4) or to four different destinations (large networks, traffic cases #5 and #6). Thus, congestion trees will be formed. When sudden congestion is used, all the congestion sources start injecting at the same time. In the case of incremental congestion, congestion sources start to inject one after one at intervals of 20 μs. Congestion sources inject also during 300 μs in the incremental configuration.

As a second scenario we will use traces. The I/O traces used in our evaluations were provided by Hewlett-Packard Labs [33]. They include all the I/O activity generated from 1/14/1999 to 2/28/1999 at the disk interface of the *cello* system. They provide information both for the requests generated by the hosts and the answers generated by the disks. As the traces are six years old, and the technology grows quickly, allowing the use of faster devices (hosts and storage devices) that generate higher injection rates, we have applied a time compression factor to the traces.

5.3 Performance Comparison

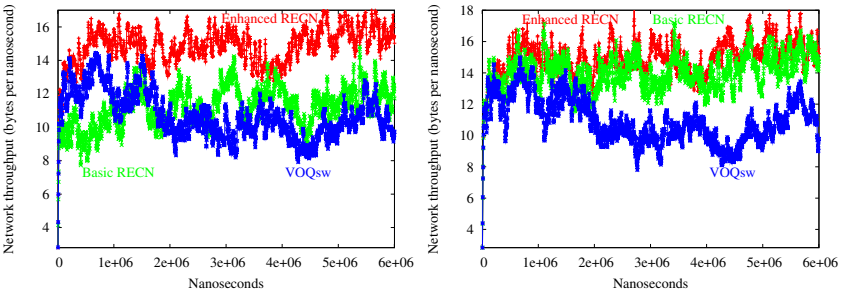
Figure 7 shows results for basic RECN, enhanced RECN and VOQsw, for the traffic cases #1 and #2. Speedup of 1.5 is used. In these two cases, the congestion tree is formed by the incremental addition of flows, and the basic RECN mechanism achieves roughly the same performance that as enhanced RECN. However, there are significant differences. First, basic RECN exhibits an oscilation in the throughput achieved, due to the fact that HOL blocking is not completely eliminated. The enhanced RECN mechanism achieves a smooth performance. In the traffic case #1, sources injecting to random destinations are injecting at half the link rate. It can be observed that when congestion tree is forming, no strong degradation is experienced. However, for traffic case #2, sources injecting random traffic are injecting at the full injection rate. It can be noticed that the basic RECN mechanism starts to degrade performance as throughput drops from 44 bytes/ns to 37 bytes/ns. However, when congestion tree disapears, it recovers, although exhibiting a significant oscillation. On the other hand, enhanced RECN behaves optimally as it tolerates the congestion tree with no performance degradation.

Taking the throughput results of VOQsw as a reference, it can be deduced that both RECN versions handle the congestion caused by traffic case #2 very well, whereas the performance achieved by VOQsw is quite poor (throughput drops from 44 to 25 bytes/ns and does not recover even when the congestion tree disappears). Therefore, we can consider that traffic case #2 can cause strong HOL blocking in the network. However, taking into account the performance of basic and enhanced RECN, we can conclude that the use of any of them virtually eliminates HOL blocking.



(a) Traffic case #1 (b) Traffic case #2

Fig. 7. Network throughput for traffic cases #1 and #2. Speedup is 1.5

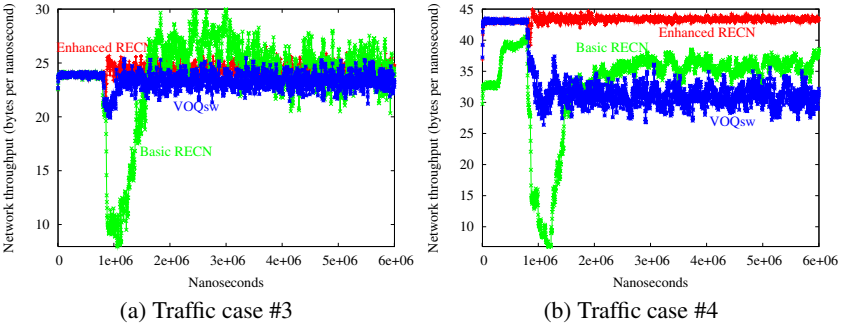


(a) No Speedup (a) Speedup 1.5

Fig. 8. Network throughput for SAN traffic

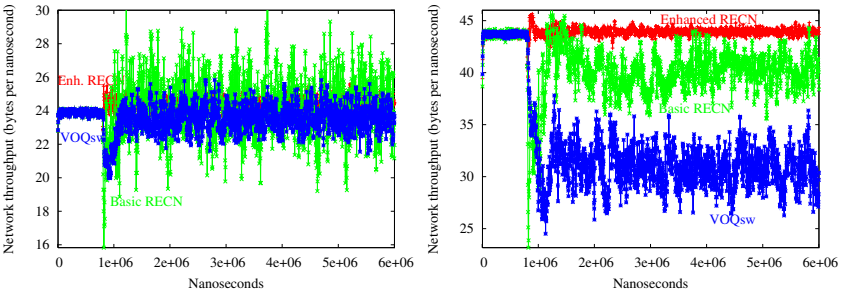
Figure 8 shows results for the SAN traces, when using no speedup (Figure 8.a) and when using 1.5 speedup (Figure 8.b). As can be noticed, the basic RECN achieves worse performance when no speedup is used, whereas the enhanced RECN works equally with or without speedup. This result shows that it completely eliminates HOL blocking at the ingress ports. When using speedup, most of the congestion is roughly moved to the egress queues and, thus, the basic RECN behaves as the enhanced RECN. Thus, when no speedup is available, enhanced RECN makes a difference.

From these two previous results (traffic cases #1 and #2 and SAN traffic) it can be deduced that basic RECN behaves acceptably with moderated traffic and when switches with speedup are available. However, these two conditions will not be always true. Figure 9 shows performance results when no speedup is available and a sudden congestion tree forms. In Figure 9.a sources injecting random traffic inject at the half of the link rate (traffic case #3) whereas in Figure 9.b they inject at the full link rate (traffic case #4). As can be observed, the basic RECN suffers strong degradation in both cases. Network throughput drops from 25 bytes/ns to 10 bytes/ns (60% drop) for traffic case #3 and from 44 bytes/ns to 10 bytes/ns (77% drop) for traffic case #4. It can be noticed also that basic RECN recovery time is quite excesive for traffic case #4 (indeed, it never



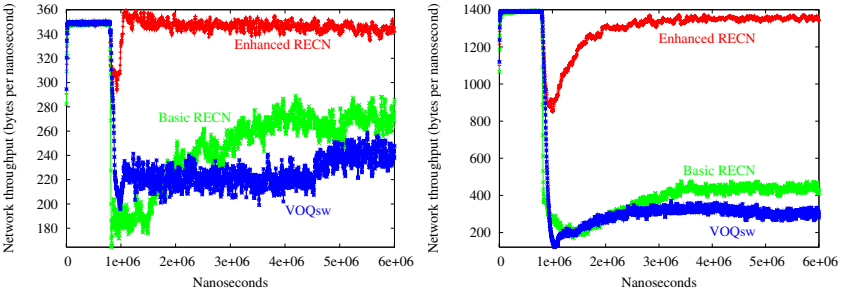
(a) Traffic case #3 (b) Traffic case #4

Fig. 9. Network Throughput for traffic cases #3 and #4. No speedup



(a) Traffic case #3 (b) Traffic case #4

Fig. 10. Network Throughput for traffic cases #3 and #4. Speedup is 1.5.



(a) Traffic case #5, 512×512 network (b) Traffic case #6, 2048×2048 network

Fig. 11. Network Throughput for traffic cases #5 and #6. Speedup is 1.5.

fully recovers). On the other hand, the enhanced RECN mechanism is able to filter the inefficiencies induced by the absence of speedup and by the dynamics of congestion trees, achieving maximum performance. Thus, it virtually eliminates HOL blocking.

Figure 10 also shows performance results for a sudden congestion tree formation, when switch speedup is 1.5. Again, network throughput for traffic cases #3 and #4 are shown in Figure 10.a and Figure 10.b, respectively. With both traffics, the behavior of the basic RECN mechanism is better than in the case of no-speedup switches, due to the fact that congestion tends to appear at egress sides. However, the basic RECN mechanism still exhibits worse performance than the enhanced one. Whereas the enhanced RECN keeps almost constantly network throughput at maximum, the basic RECN troughput drops significantly (33% for traffic case #3 and 45% for traffic case #4) and exhibits strong oscillations. Moreover, when sources inject at full rate (traffic case #4), network troughput does not completely recover from the drop. When the congestion tree disappears, the basic RECN performance is far better than the VOQsw one.

Figure 11 shows performance results for basic RECN and enhanced RECN when used on larger networks. Figure 11.a corresponds to a 512×512 MIN network (512 hosts, 640 switches), whereas Figure 11.b corresponds to a 2048×2048 MIN network (2048 hosts, 3072 switches). In both cases, switch speedup of 1.5 has been considered. Due to the big size of these networks, instead of producing just one tree, four congestion

trees are suddenly formed by sources injecting at the full link rate (traffic cases #5 and #6, respectively). It can be seen in the figures that the differences on the performance of basic and enhanced RECN grow with network size. The enhanced version keeps network throughput close to the maximum independently of network size, whereas the basic RECN throughput drops dramatically and practically does not recover.

6 Conclusions

We have analyzed the complex nature of the formation of congestion trees in networks with CIOQ switches. We have presented examples showing that, contrary to the common belief, a congestion tree can grow in a variety of ways, depending on switch architecture (crossbar speedup) and traffic load. Also, we have analyzed the importance of considering such variety when designing congestion management strategies for lossless networks, specifically those strategies focused on eliminate HOL blocking. Moreover, taking into account the previous analysis, we have proposed two significant enhancements to a recently proposed congestion management mechanism (RECN) in the aim of handling congestion trees regardless the way they form. The comparative results presented in the paper show that network performance degrades dramatically in several scenarios when using basic RECN, whereas the enhanced RECN is able to keep almost maximum performance in all the cases. So, we can conclude that the proposed enhancements allow RECN to correctly eliminate the HOL blocking produced by congestion trees independently of the way they grow.

References

1. G. S. Almasi, A. Gottlieb, "Highly parallel computing", *Ed. Benjamin-Cummings Publishing Co., Inc.*, 1994.
2. T. Anderson, S. Owicki, J. Saxe, and C. Thacker, "High-Speed Switch Scheduling for Local-Area Networks", *ACM Trans. on Computer Systems*, vol. 11, no. 4, pp. 319–352, Nov. 1993.
3. E. Baydal, P. Lopez and J. Duato, "A Congestion Control Mechanism for Wormhole Networks", in *Proc. 9th. Euromicro Workshop Parallel & Distributed Processing*, pp. 19–26, Feb. 2001.
4. E. Baydal and P. Lopez, "A Robust Mechanism for Congestion Control: INC", in *Proc. 9th International Euro-Par Conference*, pp. 958–968, Aug. 2003.
5. R. Bianchini, T. J. LeBlanc, L. I. Kontothanassis, M. E. Crovella,"Alleviating Memory Contention in Matrix Computations on Large-Scale Shared-Memory Multiprocessors", Technical report 449, Dept. of Computer Science, Rochester University, April 1993.
6. W. J. Dally, P. Carvey, and L. Dennison, "The Avici Terabit Switch/Router", in *Proc. Hot Interconnects 6*, Aug. 1998.
7. W. J. Dally, "Virtual-channel flow control", *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 194–205, March 1992.
8. W. J. Dally and H. Aoki, "Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels", *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 4, pp. 466–475, April 1993.
9. S. P. Dandamudi, "Reducing Hot-Spot Contention in Shared-Memory Multiprocessor Systems", in *IEEE Concurrency*, vol. 7, no 1, pp. 48–59, January 1999.

10. J. Duato, "A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks", *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 12, pp. 1320–1331, Dec. 1993.
11. J. Duato, S. Yalamanchili, and L. M. Ni, *Interconnection Networks: An Engineering Approach* (Revised printing), Morgan Kaufmann Publishers, 2003.
12. J. Duato, I. Johnson, J. Flich, F. Naven, P.J. Garcia, T. Nachiondo, "A New Scalable and Cost-Effective Congestion Management Strategy for Lossless Multistage Interconnection Networks", in *Proc. 11th International Symposium on High-Performance Computer Architecture* (HPCA05), pp. 108–119, Feb. 2005.
13. D. Franco, I. Garces, and E. Luque, "A New Method to Make Communication Latency Uniform: Distributed Routing Balancing", in *Proc. ACM International Conference on Supercomputing* (ICS99), pp. 210–219, May 1999.
14. P. T. Gaughan and S. Yalamanchili, "Adaptive Routing Protocols for Hypercube Interconnection Networks", *IEEE Computer*, vol. 26, no. 5, pp. 12–23, May 1993.
15. W.S.Ho, D.L.Eager, "A Novel Strategy for Controlling Hot Spot Contention", in *Proc. Int. Conf. Parallel Processing*, vol. I, pp. 14–18, 1989.
16. InfiniBand Trade Association, "InfiniBand Architecture. Specification Volume 1. Release 1.0". Available at http://www.infinibandta.com/.
17. M. Karol, M. Hluchyj, and S. Morgen, "Input versus Output Queueing on a Space Division Switch", in *IEEE Transactions on Communications*, vol. 35, no. 12, pp.1347-1356, 1987.
18. M. Katevenis, D. Serpanos, E. Spyridakis, "Credit-Flow-Controlled ATM for MP Interconnection: the ATLAS I Single-Chip ATM Switch", in *Proc. 4th Int. Symp. on High-Performance Computer Architecture*, pp. 47–56, Feb. 1998.
19. J. H. Kim, Z. Liu, and A. A. Chien, "Compressionless Routing: A Framework for Adaptive and Fault-Tolerant Routing", *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, no. 3, 1997.
20. S. Konstantinidou and L. Snyder, "Chaos Router: Architecture and Performance", in *Proc. 18th International Symposium on Computer Architecture*, pp. 79–88, June 1991.
21. V. Krishnan and D. Mayhew, "A Localized Congestion Control Mechanism for PCI Express Advanced Switching Fabrics", in *Proc. 12th IEEE Symp. on Hot Interconnects*, Aug. 2004.
22. J. Liu, K. G. Shin,C. C. Chang, "Prevention of Congestion in Packet-Switched Multistage Interconnection Networks", *IEEE Transactions on Parallel Distributed Systems*, vol. 6, no. 5, pp. 535–541, May 1995.
23. P. Lopez and J. Duato, "Deadlock-Free Adaptive Routing Algorithms for the 3D-Torus: Limitations and Solutions", in *Proc. Parallel Architectures and Languages Europe 93*, June 1993.
24. Myrinet 2000 Series Networking. Available at http://www.cspi.com/multicomputer/products/2000_series_networking/ 2000_networking.htm.
25. "Advanced Switching for the PCI Express Architecture". White paper. Available at http://www.intel.com/technology/pciex press/devnet/AdvancedSwitching.pdf
26. "Advanced Switching Core Architecture Specification". Available at http://www.asi-sig.org/specifications for ASI SIG.
27. G. Pfister and A. Norton, "Hot Spot Contention and Combining in Multistage Interconnect Networks", *IEEE Trans. on Computers*, vol. C-34, pp. 943–948, Oct. 1985.
28. Quadrics QsNet. Available at http://doc.quadrics.com
29. S. L. Scott, G. S. Sohi,"The Use of Feedback in Multiprocessors and Its Application to Tree Saturation Control", *IEEE Transactions on Parallel Distributed Systems*, vol. 1, no. 4, pp. 385–398, Oct. 1990.
30. L. Shang, L. S. Peh, and N. K. Jha, "Dynamic Voltage Scaling with Links for Power Optimization of Interconnection Networks", in *Proc. Int. Symp. on High-Performance Computer Architecture*, pp. 91–102, Feb. 2003.
31. A. Singh, W. J. Dally, B. Towles, A. K. Gupta, "Globally Adaptive Load-Balanced Routing on Tori", *Computer Architecture Letters*, vol. 3, no. 1, pp. 6–9, July 2004.

32. A. Smai and L. Thorelli, "Global Reactive Congestion Control in Multicomputer Networks", in *Proc. 5th Int. Conf. on High Performance Computing*, 1998.
33. *SSP homepage*, http://ginger.hpl.hp.com/research/itc/csl/ssp/
34. J. M. Stine and N. P. Carter, "Comparing Adaptive Routing and Dynamic Voltage Scaling for Link Power Reduction", *Computer Architecture Letters*, vol. 3, no. 1, pp. 14–17, July 2004.
35. Y. Tamir and G. L. Frazier, "Dynamically-Allocated Multi-Queue Buffers for VLSI Communication Switches", *IEEE Trans. on Computers*, vol. 41, no. 6, June 1992.
36. M. Thottethodi, A. R. Lebeck, S. S. Mukherjee, "Self-Tuned Congestion Control for Multiprocessor Networks", in *Proc. Int. Symp. High-Performance Computer Architecture*, Feb. 2001.
37. M. Thottethodi, A. R. Lebeck, S. S. Mukherjee, "BLAM: A High-Performance Routing Algorithm for Virtual Cut-Through Networks", in *Proc. Int. Parallel and Distributed Processing Symp.* (IPDPS), April 2003.
38. W. Vogels et al., "Tree-Saturation Control in the AC3 Velocity Cluster Interconnect", in *Proc. 8th Conference on Hot Interconnects*, Aug. 2000.
39. M. Wang, H. J. Siegel, M. A. Nichols, S. Abraham, "Using a Multipath Network for Reducing the Effects of Hot Spots", *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no.3, pp. 252–268, March 1995.
40. C. Q. Yang and A. V. S. Reddy, "A Taxonomy for Congestion Control Algorithms in Packet Switching Networks", *IEEE Network*, pp. 34–45, July/Aug. 1995.
41. P. Yew, N. Tzeng, D. H. Lawrie," Distributing Hot-Spot Addressing in Large-Scale Multiprocessors", *IEEE Transactions on Computers*, vol. 36, no. 4, pp. 388–395, April 1987.

A Single (Unified) Shader GPU Microarchitecture for Embedded Systems*

Victor Moya[1], Carlos González, Jordi Roca,
Agustín Fernández, and Roger Espasa[2]

Department of Computer Architecture, Universitat Politècnica de Catalunya

Abstract. We present and evaluate the TILA-rin GPU microarchitecture for embedded systems using the ATTILA GPU simulation framework. We use a trace from an execution of the Unreal Tournament 2004 PC game to evaluate and compare the performance of the proposed embedded GPU against a baseline GPU architecture for the PC. We evaluate the different elements that have been removed from the baseline GPU architecture to accommodate the architecture to the restricted power, bandwidth and area budgets of embedded systems. The unified shader architecture we present processes vertices, triangles and fragments in a single processing unit saving space and reducing hardware complexity. The proposed embedded GPU architecture sustains 20 frames per second on the selected UT 2004 trace.

1 Introduction

In the last years the embedded market has been growing at a fast peace. With the increase of the computational power of the CPUs mounted in embedded systems and the increase in the amount of available memory those systems have become open to new kind of applications. One of these applications are 3D graphic applications, mainly games. Modern PDAs, powerful mobile phones and portable consoles already implement relatively powerful GPUs and support games with similar characteristics and features of PC games from five to ten years ago. However at the current pace embedded GPUs are about to reach the programmability and performance capabilities of their 'big brothers', the PC GPUs. The last embedded GPU architectures presented by graphic companies like PowerVR [18], NVidia [19], ATI [20] or BitBoys [21] already implement vertex and pixel shaders. The embedded and mobile market are still growing and research on generic and specific purpose processors targeted to these systems will become even more important.

* This work has been supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIC2001-0995-C02-01 and TIN2004-07739-C02-01.

[1] Research work supported by the Department of Universities, Research and Society of the Generalitat de Catalunya and the European Social Fund.

[2] Intel Labs Barcelona (BSSAD), Intel Corp.

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 286 – 301, 2005.

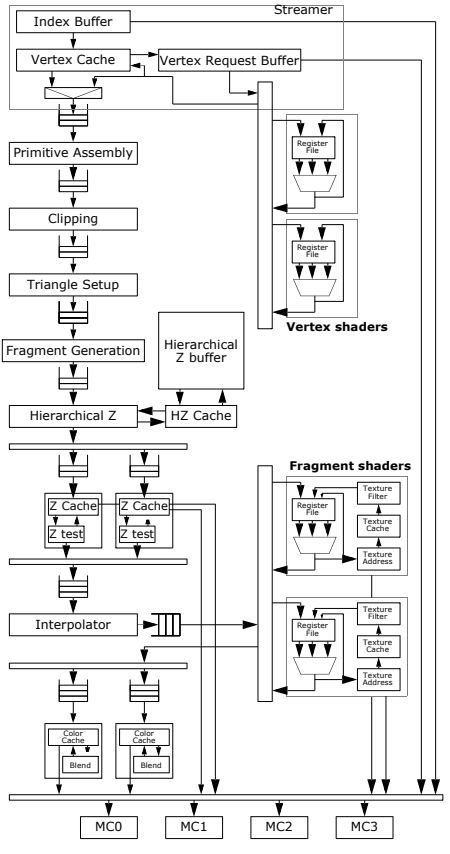


Fig. 1. ATTILA Architecture

We have developed a generic GPU microarchitecture that contains most of the advanced hardware features seen in today's major GPUs. We have liberally blended techniques from all major vendors and also from the research literature [12], producing a microarchitecture that closely tracks today's GPUs without being an exact replica of any particular product available or announced. We have then implemented this microarchitecture in full detail in a cycle-level, execution-driven simulator. In order to feed this simulator, we have also produced an OpenGL driver for our GPU and an OpenGL capture tool able to work in coordination to run full applications (i.e., commercial games) on our GPU microarchitecture. Our microarchitecture and simulator are versatile and highly configurable and can be used to evaluate multiple configurations ranging from GPUs targeted for high-end PCs to GPUs for embedded systems like small PDAs or mobile phones.

We use this capability in the present paper to evaluate a GPU microarchitecture for embedded systems and compare the performance differences from GPU architectures ranging from the middle-end PC market to the low end embedded market.

The remainder of this paper is organized as follows: Section 2 introduces the ATTILA microarchitecture and compares the baseline GPU pipeline with the proposed TILA-rin embedded GPU pipeline. Sections 3 presents the simulator itself and the associated OpenGL framework. Section 4 describes our unified shader architecture and our implementation of the triangle setup stage in the unified shader, a technique useful to reduce the transistors required for a low-end embedded GPU. Section 5 evaluates the performance of our embedded GPU architecture using a trace from the Unreal Tournament 2004 PC game and compares its performance with multiple GPU architecture configurations. Finally Sections 6 and 7 present related work and conclusions.

2 ATTILA Architecture

The rendering algorithm implemented in modern GPUs is based on the rasterization of textured shaded triangles on a color buffer, using a Z buffer to solve the visibility

problem. GPUs implement this algorithm with the pipeline shown at figure 1 and 2: the streamer stage fetches vertex data, the vertex shader processes vertices, the primitive assembly stage groups vertices into triangles, the clipper culls non visible triangles, the triangle setup and fragment generator stages create fragments from the input triangle, the z stage culls non visible fragments, the fragment shader processes those fragments and the color stage updates the framebuffer. Graphic applications can access and configure the GPU pipeline using graphic APIs like OpenGL and Direct3D.

Table 1. Bandwidth, queues and latency in cycles for the baseline ATTILA architecture

Unit	Input Bandwidth	Output Bandwidth	Input Queue		Latency in cycles
			Size	Element width	
Streamer	1 index	1 vertex	80	16×4×32 bits	Mem
Primitive Assembly	1 vertex	1 triang.	64	3×16×4×32 bits	1
Clipping	1 triang.	1 triang	32	3×4×32 bits	6
Triangle Setup	1 triang.	1 triang	32	3×4×32 bits	10
Fragment Generation	1 triang	2×64 frag.	16	3×4×32 bits	1
Hierarchical Z	2×64 frag.	2×64 frag.	64	(2×16+4×32)×4 bits	1
Z Test	4 frag.	4 frag.	64	(2×16+4×32)×4 bits	2+Mem
Interpolator	2×4 frag.	2×4 frag.	-	-	2 to 8
Color Write	4 frag.		64	(2×16+4×32)×4 bits	2+Mem
Vertex Shader	1 vertex	1 vertex	12+4	16×4×32 bits	variable
Fragment Shader	4 frag.	4 frag.	112+16	10×4×32 bits	variable

Our implementation of the pipeline correlates in most aspects with current real GPUs, except for one design decision: we decided to support from the start a unified shader model [14] in our microarchitecture. The simulator can also be configured to emulate today's hard partitioning of vertex and fragment shaders so that both models can be compared.

Table 2. Baseline ATTILA architecture caches

Cache	Size (KB)	Associativity	Lines	Line Size (bytes)	Ports
Texture	**16**	**4**	**64**	**256**	**4x4**
Z	**16**	**2**	**64**	**256**	**4**
Color	**16**	**2**	**64**	**256**	**4**

Figure 1, Table 1 and Table 2 describe our baseline non unified architecture: 4 vertex shaders, two 2-way fragment shaders, 2 ROP (raster operator) units, four 64-bit DDR channels to GPU memory, and a 2 channel bus to system memory (similar to PCI Express). Our architecture can be scaled (down or up) changing the number of

shader and ROP units and their capabilities. Table 1 shows the input and output processing elements for each pipeline stage, as well as their bandwidth, queue sizes and latencies in cycles. Table 2 shows the configuration of the different caches in the pipeline. Figure 2 shows the TILA-rin embedded GPU. The embedded architecture is configured with a single unified shader and ROP unit. A more detailed description of the ATTILA pipeline and stages can be found at [23]. In this section we will only compare the base line architecture and the proposed embedded architecture.

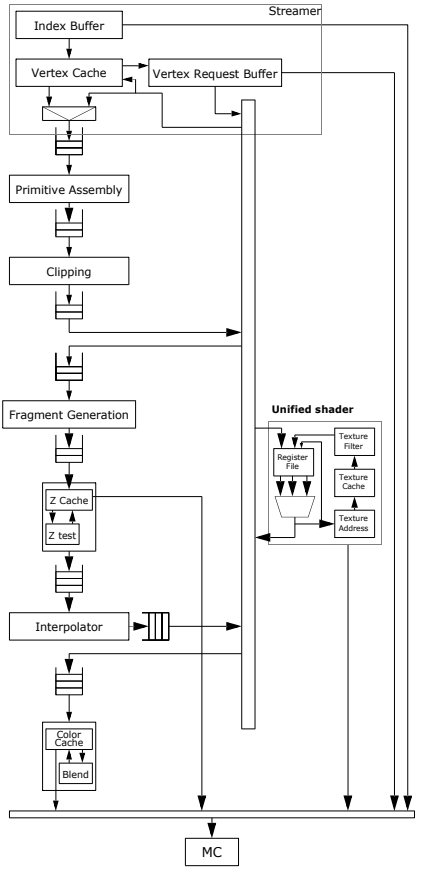


Fig. 2. TILA-rin unified shader

Comparing Figures 1 and 2 we can see the main differences. The baseline PC GPU configuration implements separated vertex (four, but only two shown in Figure 1) and fragment shader units while the TILArin embedded GPU implements a single unified shader. This single unified shader processes three kind of inputs: vertices, triangles and fragments.

The Triangle Setup stage is mostly removed (further discussed at section 4.1) from the embedded GPU and the triangle setup computation is performed in the single shader unit. The on die Hierarchical Z buffer [10][11] and associated fast fragment cull stage are also removed in the embedded GPU to reduce the transistor budget.

The embedded architecture implements a single ROP pipeline (z test, stencil test and color write), two in the baseline architecture, and a single channel to GPU memory (16 bytes per cycle) compared with four in the baseline architecture. The bus to system memory is the same for both configurations. Z compression isn't supported in the embedded architecture and the Z and color cache are smaller, 4 KB, with the cache line size set at 64 bytes rather than 256 bytes (Table 2).

The ROP stages also differ in the rate at which fragments are processed. While the embedded architecture keeps a quad (2x2 fragments) as the work unit a reduced number of ALUs, internal bus widths and band width limit the throughput to one fragment per cycle. The baseline PC ROP units can process a whole quad in parallel and output one quad per cycle.

The baseline fragment shader unit can also process a whole quad in parallel while the vertex shader and unified shader only work on a single processing element per cycle.

3 ATTILA Simulator and OpenGL Framework

We have developed a highly accurate, cycle-level and execution driven simulator for the ATTILA architecture described in the previous section. The model is highly configurable (over 100 parameters) and modular, to enable fast yet accurate exploration of microarchitectural alternatives.

The simulator is "execution driven" in the sense that real data travels through buses between the simulated stages. A stage uses the data received from its input buses and the data it stores on its local structures to call the associated functional module that creates new or modified data that continues flowing through the pipeline. The same (or equivalent) accesses to memory, hits and misses and bandwidth usage that a real GPU are generated. This key feature of our simulator allows to verify that the architecture is performing the expected tasks.

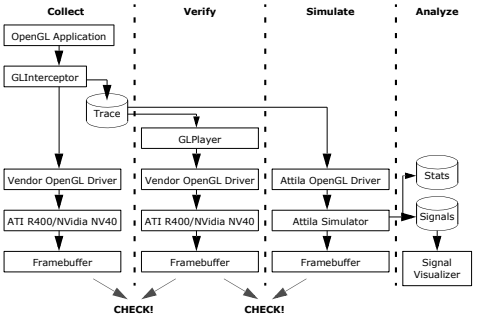


Fig. 3. ATTILA Simulation Framework

Our simulator implements a "hot start" technique that allows the simulation to be started at any frame of a trace file. Frames, disregarding data preloaded in memory, are mostly independent from each other and groups of frames can be simulated independently. A PC cluster with 80 nodes is used to simulate dozens of frames in parallel. The current implementation of the simulator can simulate up to 50 frames at 1024x768 of a UT2004 trace, equivalent to 200-300 million cycles, in 24 hours in a single node (P4 Xeon @ 2 GHz).

We have developed an OpenGL framework (trace capturer, library and driver) for our ATTILA architecture (D3D is in the works). Figure 3 shows the whole framework and the process of collecting traces from real graphic applications, verifying the trace, simulating the trace and verifying the simulation result.

Our OpenGL stack bridges the gap between the OpenGL API and the ATTILA architecture. The stack top layer manages the API state while the lower layer offers a basic interface for configuring the hardware and allocating memory. The current implementation supports all the API features required to fully simulate a graphic application. One key feature is the conversion of the vertex and fragment fixed function API calls into shader programs [13], required applications using the pre programmable shading API.

The GLInterceptor tool uses an OpenGL stub library to capture a trace of all the OpenGL API calls and data that the graphic application is generating as shown in Figure 3. All this information is stored in an output file (a trace). To verify the integrity and faithfulness of the recorded trace the GLPlayer can be used to reproduce the trace.

After the trace is validated, it is feed by our OpenGL stack into the simulator. Using traces from graphic applications isolates our simulator from any non GPU

system related effects (for example CPU limited executions, disk accesses, memory swapping). Our simulator uses the simulated DAC unit to dump the rendered frames into files. The dumped frame is used for verifying the correctness of our simulation and architecture.

4 Unified Shader

Our unified shader architecture is based on the ISA described at the ARB vertex and fragment program OpenGL extensions [15][16]. The shader works on 4 component 32 bit float point registers. SIMD operations like addition, dot product and multiply-add are supported. Scalar instructions (for example reciprocate, reciprocate square root, exponentiation) operating on a single component of a shader register are also implemented. For the fragment shader target and for our unified shader target texture, instructions for accessing memory and a kill instruction for culling the fragment are supported.

The ARB ISA defines four register banks: a read only register bank for the shader input attributes, a write only register bank for the shader output attributes, a read-write temporal register bank to store intermediate results and a read only constant bank that stores data that doesn't change per input but per batch and is shared by all the shader processors. Figure 4 shows the unified shader programming environment.

Shader instructions are stored in a relatively small shader instruction memory (the ARB specification requires a memory for at least 96 instructions) that is preloaded before the batch rendering is started. The shader processor pipeline has a fetch stage, a decode stage, an instruction dependant number of execution stages and a write back stage. The shader processor executes instructions in order and stalls when data dependences are detected. The instruction execution latencies range from 1 cycle to 9 cycles for the arithmetic instructions in our current implementation but can be easily configured in the simulator to evaluate more or less aggressively pipelined ALUs.

Our shader architecture implements multithreading to hide instruction execution latencies and texture access latency exploiting the inherent parallelism of the shader inputs. From the shader point of view all shader inputs are completely independent. A thread represents a shader input. For the vertex shader target only a few threads are required to hide the latency between dependant instructions, so for our baseline implementation only 12 threads are supported. For the fragment shader and unified shader targets more threads are required to hide the latency of the texture accesses therefore we support up to 112 shader inputs on execution in our baseline configuration. A texture access blocks the thread until the texture operation finishes preventing the fetch stage from issuing instructions for that thread.

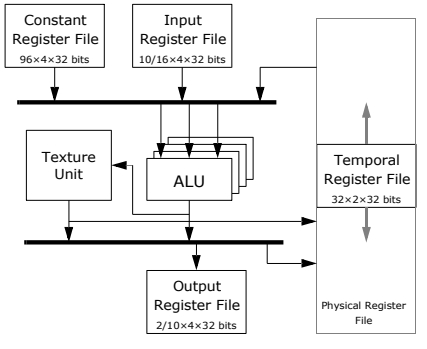


Fig. 4. Unified shader architecture

The number of threads on execution is further limited by the maximum number of temporal live registers that the running shader program requires. The ARB ISA defines up to 32 registers in the temporal register bank but less temporal registers are required for the applications that we have tested. We provide a pool of 96 physical registers for the vertex shader target and 448 physical registers for the fragment and unified shader targets. The vertex programs that we have analyzed until now require 4 to 8 temporal registers per program (vertex programs may have up to a hundred instructions when implementing multiple light sources and complex lighting modes) while the fragment programs require 2 to 4 registers (fragment programs feature no more than a dozen instructions in most analyzed cases).

Another characteristic of our shader model is that we support, for the fragment and unified shader target, to work on groups of shader inputs as a single processing unit. This is a requirement for our current texture access implementation as explained in [23]. The same instructions are fetched, decoded and executed for a group of four inputs. For the baseline architecture the four inputs that form a group are processed in parallel and the shader unit works as a 512 bit processor (4 inputs, 4 components, 32 bits per component). For the embedded GPU configurations the shader unit executes the same instructions in sequence for one (128 bit wide shader processor) or two inputs (256 bit wide shader processor) from a group until the whole input group is executed. Our fetch stage can be configured to issue one or more instructions for an input group per cycle. Our baseline architecture can fetch and start two instructions for a group per cycle while the embedded architecture can only fetch and start one instruction per cycle. For the vertex shader target each input is treated independently as vertex shaders in current GPUs are reported to work [1].

4.1 Triangle Setup in the Shader

The triangle rasterization algorithm that our architecture implements (based on [8][9]) targets the triangle setup stage for an efficient implementation on a general purpose CPU [9] or in a separated geometric processor [8]. Our baseline architecture implements the triangle setup stage as a fixed function stage using a number of SIMD ALUs. But the setup algorithm is also suited for being implemented using a shader program.

The triangle setup algorithm implemented can be divided in a processing stage that is the part that will be implemented in the unified shader unit (Figure 2) and a small triangle culling stage that will remain as a separated fixed function stage. Figure 5 shows the shader program used to perform the setup computation for an input triangle.

The processing stage takes the 2D homogenous coordinate vectors (x, y and w components) for the three vertices that form the triangle and creates an input 3x3 matrix M. Then the adjoint matrix for the input matrix - A = adj (M) - is calculated (lines 1 - 6). The determinant of the input matrix - det(M) - is also calculated (lines 7 - 8). If the determinant is 0 the triangle doesn't generate any fragment and can be culled. The next step of the algorithm described in [8][9] calculates the inverse matrix - M^{-1} = adj(M)/ det(M) - but that step isn't always required and we don't implement it. The resulting setup matrix rows are the three triangle half plane edge equations that are used in the Fragment Generation stage to generate the triangle fragments. The

resulting setup matrix can be used to create an interpolation equation to perform linear perspective corrected interpolation of the fragment attributes from the corresponding triangle vertex attributes.

```
#  Define input attributes
#  Definitions
#
#  Define input attributes
#

ATTRIB iX = triangle.attrib[0];
ATTRIB iY = triangle.attrib[1];
ATTRIB iZ = triangle.attrib[2];
ATTRIB iW = triangle.attrib[3];

#
#  Define the constant parameters
#

#
#  Viewport parameters (constant per batch)
#
#       (x0, y0) : viewport start position
#       (width, height)   : viewport size
#

PARAM rFactor1 = 2.0 / width;
PARAM rFactor2 = 2.0 / height;
PARAM rFactor3 = -(((x0 * 2.0) / width) + 1);
PARAM rFactor4 = -(((y0 * 2.0) / height) + 1);
PARAM offset = 0.5;

#
#  Define output attributes
#

OUTPUT oA = result.aCoefficient;
OUTPUT oB = result.bCoefficient;
OUTPUT oC = result.cCoefficient;
OUTPUT oArea = result.color.back.primary;


#  Define temporal registers
#

TEMP rA, rB, rC, rArea, rT1, rT2;


#  Code

#
#  Calculate setup matrix (edge equations) as
#  the adjoint of the input vertex position
#  matrix.
#

1:  MUL rC.xyz, iX.zxyw, iY.yzxw;
2:  MUL rB.xyz, iX.yzxw, iW.zxyw;
3:  MUL rA.xyz, iY.zxyw, iW.yzxw;
4:  MAD rC.xyz, iX.yzxw, iY.zxyw, -rC;
5:  MAD rB.xyz, iX.zxyw, iW.yzxw, -rB;
6:  MAD rA.xyz, iY.yzxw, iW.zxyw, -rA;
#
#  Calculate the determinant of the input
#  matrix (estimation of the signed 'area'
#  for the triangle).
#

7:  DP3 rArea, rC, iW;
8:  RCP rT2, rArea.x;

#
# Calculate Z interpolation equation.
#

9:  DP3 rA.w, rA, iZ;
10:  DP3 rB.w, rB, iZ;
11:  DP3 rC.w, rC, iZ;

#
#  Apply viewport transformation to equations.
#

12:  MUL rT1, rA, rFactor3;
13:  MAD rT1, rB, rFactor4, rT1;
14:  ADD rC, rC, rT1;
15:  MUL rA, rA, rFactor1;
16:  MUL rB, rB, rFactor2;
17:  MUL rA.w, rA.w, rT2.x;
18:  MUL rB.w, rB.w, rT2.x;
19:  MUL rC.w, rC.w, rT2.x;

#
#  Apply half pixel sample point offset
#  (OpenGL).
#

20:  MUL rT1, rA, offset;
21:  MAD rT1, rB, offset, rT1;
22:  ADD rC, rC, rT1;

#
#  Write output registers.
#

23:  MOV oA, rA;
24:  MOV oB, rB;
25:  MOV oC, rC;
26:  MOV oArea.x, rArea.x;
27:  END;
```

Fig. 5. Shader program for Triangle Setup

The interpolation equation is computed by a vector matrix product between a vector storing the attribute value for the three vertices and the calculated setup matrix (lines 9 - 11). We only calculate the interpolation equation for the fragment depth

attribute as it is the only attribute required for fast fragment depth culling and is automatically generated for each fragment in the Fragment Generator stage. The other fragment attributes will be calculated later in the Interpolation unit using the barycentric coordinates method.

The last step of the processing stage is to adjust the four equations to the viewport size and apply the OpenGL half pixel sampling point offset (lines 12 - 19 and 20 - 22). The result of the processing stage are three 4 component vectors storing the three coefficients (a, b, c) of the triangle half plane edge and z interpolation equations and a single component result storing the determinant of the input matrix to be used for face culling (lines 23 - 26).

The triangle culling stage culls triangles based on their facing and the current face culling configuration. The sign of the input matrix (M) determinant determines the facing direction of the triangle. Front facing triangles (for the counter-clock-wise vertex order default for OpenGL) have positive determinants and back facing triangles negative. If front faces are configured to pass triangles are culled when the determinant is negative, if back faces are configured to pass triangles are culled when the determinant is positive. As the Fragment Generator requires front facing edge and z equations, the equation coefficients of the back facing triangle are negated becoming a front facing triangle for the Fragment Generator.

Table 3. Evaluated GPU architecture configurations

Conf	Res	MHz	VSh	(F)Sh	Setup	Fetch way	Reg Thread	Buses	Cache	eDRAM	H Z	Compr
A	1024x768	400	4	2x4	fixed	2	4x112	4	16 KB	-	yes	yes
B	320x240	400	4	2x4	fixed	2	4x112	4	16 KB	-	yes	yes
C	320x240	400	2	1x4	fixed	2	4x112	2	16 KB	-	yes	yes
D	320x240	400	2	1x4	fixed	2	4x112	2	8 KB	-	no	yes
E	320x240	200	-	2	fixed	2	2x112	1	8 KB	-	no	yes
F	320x240	200	-	2	fixed	2	2x112	1	4 KB	-	no	no
G	320x240	200	-	1	fixed	2	4x60	1	4 KB	-	no	no
H	320x240	200	-	1	fixed	1	4x60	1	4 KB	-	no	no
I	320x240	200	-	1	on shader	1	4x60	1	4 KB	-	no	no
J	320x240	200	-	1	on shader	1	4x60	1	4 KB	1 MB	no	no
K	320x240	200	-	1	on shader	1	4x60	1	4 KB	1 MB	yes	yes

5 Embedded GPU Evaluation

Table 3 shows the parameters for the different configurations we have tested, ranging from a middle level PC GPU to our embedded TILA-rin GPU. The purpose of the different configurations is to evaluate how the performance is affected by reducing the different hardware resources. A configuration could then be selected based on the transistor, area, power and memory budgets for specific target systems. The first configuration shows the expected performance and framerate of the UT2004 game in a middle-end PC at a common PC resolution and is used as a baseline to compare the

performance of the other configurations. When we reduce the resolution to the typical of embedded systems the same base configuration shows a very high and unrealistic framerate (the game would become CPU limited) but it's useful as a direct comparison between the powerful PC configurations and the limited embedded configurations. PC CRT and high-end TFT screens supporting high refresh rates (100+ Hz) are quite common while the small LCD displays of portable and embedded systems only support screen refresh rates in the order of 30 Hz, therefore a game framerate of 20 to 30 is acceptable in the embedded world.

The tests cases A and B correspond to the same configuration, equivalent to an ATI Radeon 9800 (R350) or ATI Radeon X700 (RV410) graphic cards, while the configuration C could correspond to an ATI Radeon X300 (RV370). The other configurations range from the equivalent of a GPU integrated in a PC chipset or a high end PDA to a low end embedded GPU. Configurations A to I implemente 128 MBs of GPU memory (UT2004 requires more than 64 MBs for textures and buffers) and 64 MBs of higher latency system memory. Configurations J and K are configured with 1 MB of GPU memory for the framebuffer (could be implemented as embedded DRAM), with a maximum data rate of 16 bytes/cycle, and 128 MBs of higher latency lower bandwidth system memory.

Table 3 shows the following parameters: resolution (Res) at which the trace was simulated, an estimated working frequency in MHz for the architecture (400+ MHz is common for the middle and high-end PC GPU segments, 200 MHz is in the middle to high-end segment of the current low power embedded GPUs); the number of vertex shaders for non unified shader configurations (VS) and the number of fragment shaders (FSh) for non unified shader configurations or the number of unified shaders (Sh) for unified shader configurations. The number of fragment shaders is specified by the number of fragment shader units multiplied by the number of fragments in a group (always 4). For the unified shader configurations the number represents how many shader inputs (vertices, triangles or fragments) from a four input group are processed in parallel. The Setup parameter indicates if the Triangle Setup processing stage is performed in a separated fixed function ALU or in the shader. The Fetch way parameter is the number of instructions that can be fetched and executed per cycle for a shader input or group. The registers per thread parameter (Regs/Thread) represents the number of threads and temporal registers per thread for each shader unit in the GPU. The number of 16 bytes/cycle channels used to access the GPU memory (or the embedded memory for configurations J and K) can be found in the column labeled Buses. The size of the caches in the GPU is defined by the Caches parameter and the eDRAM parameter defines how many embedded DRAM or reserved SRAM (on the same die or in a separate die) is available for storing the framebuffer. Finally the HZ and Compr. parameters show if the configuration implements Hierarchical Z and Z compression (when disabled Z and color cache line size becomes 64 bytes).

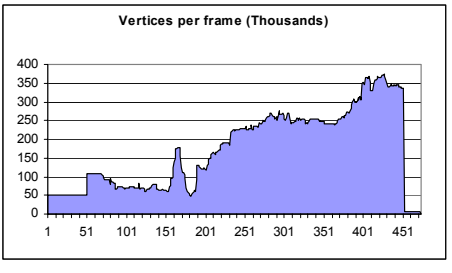


Fig. 6. Vertices per frame for the UT2004 trace

We selected the different configurations reducing, from the baseline configuration A, the number of shader and ROP pipelines and, at the same time, reducing the peak bandwidth as it exceeds the configuration requeriments. We reduce therefore the cost in area and the cost of the memory. We reduce the frequency to reduce power (the required voltage also goes down with the frequency). We reduce the cache sizes when the ratio of cache area versus ALU area changes. After reaching the single unified shader configuration (G) we keep reducing the area requeriment, first removing the second SIMD ALU (H) from the shader, then removing the triangle setup unit and performing setup in the shader (I). We finally test the impact of keeping the texture data in system memory (higher latency and half the bandwidth than the GPU memory) while using 1 MB of GPU memory for the framebuffer (J). For this configuration we also test the effect of implementing bandwidth saving techniques: Hierarchical Z and Z compression (K).

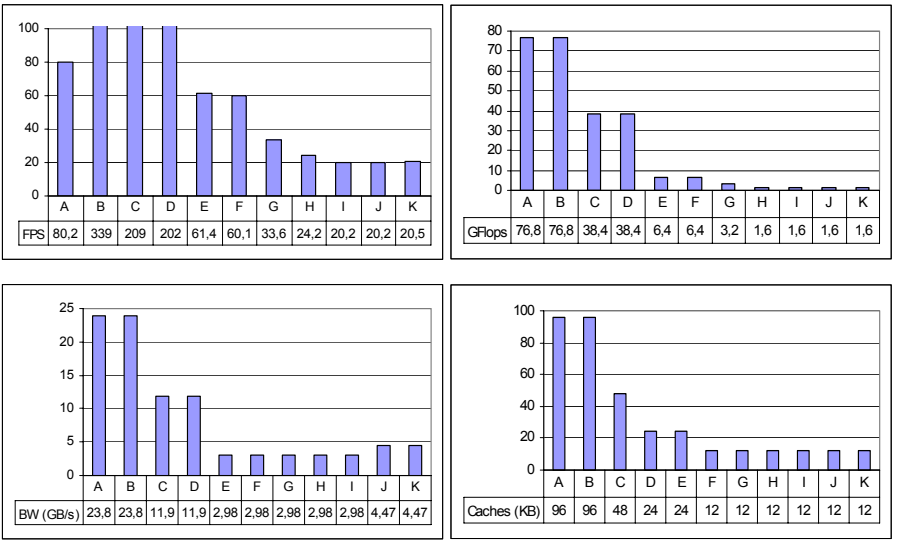


Fig. 7. FPS, peak bandwidth, peak shading power and total cache memory for all the GPU configurations

5.1 Benchmark Description

We have selected a 450 frame trace from a timedemo of the Unreal Tournament 2004 Primeval map for the evaluation of our embedded architecture. UT2004 is a game that uses a version of the Unreal game engine supporting both the OpenGL and Direct3D APIs. The UT2004 version of the Unreal engine uses the fixed function vertex and fragment OpenGL API, and our OpenGL implementation generates, as is done for real GPUs, our own shader programs from the fixed function API calls, as discussed in sec-

tion 3. The generated vertex programs are relatively large (20 to 90 instructions) while the fragment programs are mostly texture accesses with a few instructions combining the colors and performing alpha test (6 to 15 instructions). However the vertex and texture load of UT2004 is comparable (or even higher) to other current games.

We don't currently support tracing OpenGL ES (OpenGL for embedded and mobile systems) applications so we use a trace from a PC game. The workload (texture sizes, memory used and vertex load) of the selected Unreal trace, even at lower resolutions, is quite heavyweight compared with current embedded graphic applications but we consider that is a good approximation for future embedded graphic applications. The trace was generated at a resolution of 800x600 in a Pentium4 at 2.8 GHz PC with a GeForce 5900. For the experiments we modified the trace resolution (OpenGL call glViewport()) to use the 1024x768 (PC) and 320x240 (embedded) resolutions. Figure 6 shows the vertex load per frame for the selected trace. We didn't simulate the whole trace but six representative regions: frames 100 to 120, 200 to 220, 250 to 270, 300 to 320, 350 to 370 and 400 to 440. For configuration A (highest workload) the simulation lasted 7 to 14 hours for, depending on the region, 100 to 240 million simulated cycles. For the other configurations the simulation lasted from 1 to 7 hours for 150 to 500 million cycles depending on the configuration and simulated region. The simulations were performed on an 80 node cluster of P4 Xeon @ 2 GHz.

5.2 Performance

Figure 7 shows the simulated framerate (frames per second or FPS), peak bandwidth (GB/s), peak shader GFlops and the total amount of cache memory for each configuration. Figure 8 shows the relation between peak bandwidth, peak computation and the total cache memory and performance (measured in frames per second) normalized to test case A.

Figure 8 shows that the efficiency of shader processing (GFlops) decreases (configurations H to A) as the number of shader units and ALUs per shader unit increases. These configurations are likely limited by the memory subsystem. Our

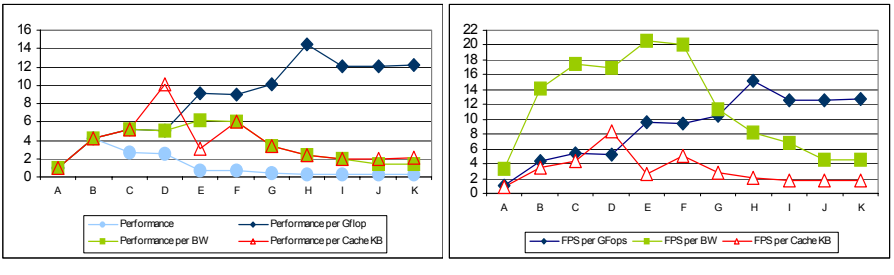


Fig. 8. The graphic in the left shows the relative performance per unit resource (1 GFlop, 1 GB/s and 1 KB). The graphic in the right shows the frames per second for the same resources.

current implementation of the Memory Controller isn't efficient enough to provide the shader and other stages of the GPU pipeline with enough data with the configured bandwidth. Improving and increasing the accuracy of the simulated memory model will be a key element of our future research. Another source of inefficiency for the 2-way configurations is the reduced ILP of the shader programs used in the trace.

Comparing, in Figure 7, configuration I against configurations J and K there is no noticeable performance hit which seems to indicate that for configuration I GPU memory bandwidth exceeds the shader processing and latency hiding capacity. Implementing a lower cost memory (configurations J and K or even less expensive configurations) makes sense for this embedded configuration. In Figure 8 we confirm that the efficiency per GB/s of bandwidth provided is higher for configurations E and F and becomes worse for all other configurations, signaling that the configured shader capacity and memory bandwidth may not optimal for the selected trace. Other graphic applications (less limited by shader processing or using bandwidth consuming techniques, for example antialiasing) may show different shading to bandwidth requirements.

Enabling Hierarchical Z and framebuffer compression in configuration K has no noticeable effect on performance. The reason is that the 1 MB framebuffer memory provides more than enough bandwidth for the framebuffer accesses and bandwidth saving techniques are not useful.

5.3 Triangle Setup in Shader Performance Analysis

To test the overhead of removing the fixed function unit that performs the processing stage of triangle setup we use five different traces. The torus trace renders four colored torus lit by a single light source. The trace has a heavy vertex and triangle load and a limited fragment load (minimum fragment shader used). The lit spheres trace renders four spheres with multiple light sources active. The vertex and triangle workload is high, the vertex shader program is large and the fragment shader program is the minimum (copy interpolated color). The space ship trace renders a multitextured space ship for a high vertex workload and small vertex and fragment shader programs. The trace may be limited by vertex data reading. The VL-II trace [17] renders a room with a volumetric lighting effect implemented with a 16 instruction fragment shader. The vertex and triangle load for this trace is low. The fifth trace is the Unreal Tournament 2004 trace (same simulated regions) that we have used in the previous section.

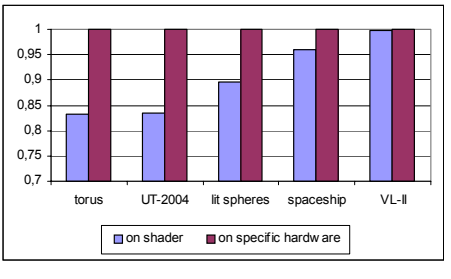


Fig. 9. Cost of performing triangle setup in the shader

Figure 9 shows the relative performance of configurations H (triangle setup in specific hardware) and I (triangle setup in the shader unit) for the five selected traces. The overhead of performing triangle setup in the shader increases as the triangle load (usually at a ratio 1:1 to 3:1 with the vertex load, depending on the rendering primitive and post shading vertex cache hits) increases in relation with the fragment load. Figure 9 shows that for the torus

trace there is a 16% hit on performance when performing triangle setup in the single shader unit. For a fragment limited application like the VL-II demo the performance reduction is too small to be noticeable. In mixed workload applications or applications limited by other stages (vertex shading for lit spheres and vertex data bandwidth for the space ship) the performance hit is reduced. The UT2004 game is a more complex graphic application and we can find frames and batches that are fragment limited and others that have a high vertex/triangle load (see Figure 6). But for the tested rendering resolution, 320x240, UT2004 has a relatively high vertex and triangle workload on average and we can see that the performance hit is quite noticeable: a 16% performance reduction. From the results in the previous subsection we also know that the UT2004 trace is limited by shader processing in both configurations and increasing the shader load, shading triangles, helps to increase the performance hit.

Performing triangle setup on the shader will make sense when the transistor budget for the GPU architecture is low and removing the large fixed function ALU can save area and power, while keeping an acceptable performance for the targeted applications. The embedded GPU configuration is clearly limited by shading performance (only one shader unit executing a single instruction per cycle). Performing Triangle Setup in the shader may be also useful when shading performance is so high that the performance overhead becomes small, in the order of a 1% reduction, for example in a PC unified architecture with 8+ shader units.

6 Related Work

A complete picture of the architecture of a modern GPU is difficult to acquire just from the regular literature. However NVidia presented their first implementation of a vertex shader for the GeForce3 (NV2x) GPU [1] and information from available patents [2], even if limited, complemented with the analysis of the performance of the shader units [24][25] in current GPUs provides an useful insight in the architecture of a modern GPUs. More actualized information about NVidia and ATI implementations surfaces as unofficial or unconfirmed information on Internet forums [3][4]. Outside shader microarchitecture some recent works can be found. For example, T. Aila et al. proposed delay streams [5], as a way to improve the performance of immediate rendering GPU architectures with minor pipeline modifications and at the Graphics Hardware 2004 conference a simulation framework for GPU simulation, QSilver [7], was presented.

Research papers and architecture presentations about embedded GPU architectures have become common in the last years. Akenine-Möller described a graphic rasterizer for mobile phones [6] with some interesting clues about how the graphic pipeline is modified to take into account low power and low bandwidth, while keeping an acceptable performance and rendering quality. Bitboys [21] and Falanx [22], both small companies working on embedded GPU architectures, presented their future architectures and their vision of the importance of embedded GPU market in Hot3D presentations in the Graphics Hardware 2004 conference.

7 Conclusions

Our flexible framework for GPU microarchitecture simulation allows to accurately model and evaluate a range of GPU configurations from a middle-end PC GPU to a low-end embedded GPU. We have presented a low budget embedded GPU with a single unified shader that performs the triangle setup computations in the shader, saving area. We have evaluated the performance of the different configurations ranging from a baseline middle-end PC GPU architecture to the proposed embedded GPU architecture. We have also evaluated the cost of performing triangle setup in the unified shader unit for different graphic traces.

The proposed embedded architecture achieves with a single unified shader unit and ROP pipe, keeping with the low power, area and memory requeriments, a sustainable rate of 20 frames per second with the selected Unreal Tournament trace at a resolution of 320x240.

References

[1] Erik Lindholm, et al. An User Programmable Vertex Engine. ACM SIGGRAPH 2001.
[2] WO02103638: Programmable Pixel Shading Architecture, December 27, 2002, NVIDIA CORP.
[3] Beyond3D Graphic Hardware and Technical Forums. http://www.beyond3d.com
[4] DIRECTXDEV mail list. http://discuss.microsoft.com/archives/directxdev.html
[5] T. Aila, V. Miettinen and P. Nordlund. Delay streams for graphics hardware. ACM Transactions on Graphics, 2003.
[6] T. Akenine-Möller and J. Ström Graphics for the masses: a hardware rasterization architecture for mobile phones. ACM Transaction on Graphics, 2003.
[7] J. W. Sheaffer, et al. A Flexible Simulation Framework for Graphics Architectures. Graphics Hardware 2004.
[8] Marc Olano, Trey Greer. Triangle Scan Conversion using 2D Homogeneous Coordinates. Graphics Hardware, 2000.
[9] Michael D. McCool, et al. Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. Proceedings Graphics Hardware 2001.
[10] Green, N. et al. Hierarchical Z-Buffer Visibility. Proceedings of SIGGRAPH 1993.
[11] S. Morein. ATI Radeon Hyper-z Technology. In Hot3D Proceedings - Graphics Hardware Workshop, 2000.
[12] Stanford University CS488a Fall 2001 Real-Time Graphics Architecture. Kurt Akeley, Path Hanrahan.
[13] Lars Ivar Igesund, Mads Henrik Stavang. Fixed function pipeline using vertex programs. November 22. 2002
[14] Microsoft Meltdown 2003, DirectX Next Slides. http://www.microsoft.com/downloads/details.aspx?FamilyId=3319E8DA-6438-4F05-8B3D-B51083DC25E6&displaylang=en
[15] ARB Vertex Program extension: http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt
[16] ARB Fragment Program extension: http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt.
[17] Volumetric Lighting II. Humus 3D demos: http://www.humus.ca/
[18] PowerVR MBX http://www.powervr.com/Products/Graphics/MBX/Index.asp

[19] ATI Imageon 2300 http://www.ati.com/products/imageon2300/features.html
[20] NVidia GoForce 4800 http://www.nvidia.com/page/goforce_3d_4500.html
[21] Bitboys G40 Embedded Graphic Processor. Hot3D presentations, Graphics Hardware 2004.
[22] Falanx Microsystems. Image Quality no Compromise. Hot3D presentations, Graphics Hardware 2004.
[23] Victor Moya , Carlos Gonzalez, Jordi Roca, Agustin Fernandez, Roger Espasa. Shader Performance Analysis on a Modern GPU Architecture. IEEE Micro-38 2005.
[24] GPUBench. http://graphics.stanford.edu/projects/gpubench/
[25] Kayvon Fatahalian, Jeremy Sugerman, Pat Hanrahan. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. Graphics Hardware 2004.

A Low-Power DSP-Enhanced 32-Bit EISC Processor

Hyun-Gyu Kim[1,2] and Hyeong-Cheol Oh[3]

[1] Dept. of Elec. and Info. Eng., Graduate School, Korea Univ., Seoul 136-701, Korea
[2] R&D center, Advanced Digital Chips, Seoul 135-508, Korea
babyworm@gmail.com
[3] Dept. of Info. Eng., Korea Univ. at Seo-Chang. Chung-Nam 339-700, Korea
ohyeong@korea.ac.kr

Abstract. EISC (Extendable Instruction Set Computer) is a compressed code architecture developed for embedded applications and has higher code density than its competing architectures. In this paper, we propose a low-power DSP-enhanced embedded microprocessor based on the 32-bit EISC architecture. We present how we could exploit the special features, and how we could overcome the deficits, of the EISC architecture to accelerate DSP applications while adding relatively low hardware overhead. Our simulation results show that the proposed DSP-enhanced processor reduces the execution time of the considered DSP kernels by 77.6% and the MP3 applications by 30.9%. The proposed DSP enhancements cost approximately 10300 gates (18%) and do not increase the clock frequency. While the high code density of EISC would be of great advantage to a low-power embedded system, the proposed DSP enhancement could increase its power consumption by 16.9%. We show that a set of supports for power management could reduce the power consumption by 65.5%. The proposed processor has been embedded in an SoC for video processing and proven in silicon.

1 Introduction

As more and more multimedia and DSP applications run on embedded systems in recent years, it has become one of the most important tasks for embedded microprocessors to accelerate DSP applications. This trend is being reflected on the most successful embedded processors in the market: ARM cores added saturation arithmetic in ARMv5TE and SIMD (Single Instruction Multiple Data) instruction set in ARMv6 [1]; and MIPS cores adopted an application-specific-extension instruction set for DSP and 3D applications [2]. These acceleration capabilities should be implemented with as little hardware overhead as possible, since most embedded microprocessors target on the cost and power sensitive markets.

In this paper, we introduce a DSP-enhanced embedded microprocessor based on the EISC (Extendable Instruction Set Computer) architecture [3,4,5]. EISC is a compressed code architecture developed for embedded applications. While

T. Conte et al. (Eds.): HiPEAC 2005, LNCS 3793, pp. 302–316, 2005.

achieving high code density and a low memory access rate, the EISC architecture uses some special features to resolve the problem of insufficient immediate operand fields [5]. We present how we could exploit the special features of EISC architecture to accelerate DSP applications while adding relatively low hardware overhead. Since EISC also has deficits in processing DSP applications like any other compressed code architectures, we propose various schemes to overcome these deficits of EISC.

In order to seek proper enhancements, we first analyzed the workload of benchmark programs from Mediabench [6] on the 32-bit EISC architecture. Based on the profiling results, we modify the base processor by carefully choosing and adding DSP instructions that can accelerate the multimedia and filtering applications. As the base processor, we use a prototype processor, called *base* below, which implements the non-DSP 32-bit instruction set of EISC [5]. We add the SIMD instructions that are selected by considering their (hardware) implementation complexity, to the processor *base*. We also adopt various schemes for signal processing, such as saturation arithmetic, a support for correcting radix point during the multiplication of fixed-point numbers, an address generation unit for supporting the branch operation efficiently, and a scheme to increase the effective number of general-purpose registers (GPRs).

Our simulation results show that the proposed DSP-enhanced processor reduces the execution time of the DSP kernels considered in this work by 77.6% and the MP3 applications by 30.9%. The proposed DSP enhancements cost approximately 10300 gates only and do not increase the clock frequency.

The high code density of EISC is of great advantage to a low-power embedded system. While the DSP enhancements we adopt increase the code density further, the hardware overhead involved in the enhancements increase the power consumption of EISC. We add the supports for static and dynamic power managements to the proposed processor. Simulation results show that the supports for power managements can reduce the power consumption of the processor by 65.5%, while the DSP enhancements increase it by 16.9%. We also introduce a power management scheme for SoCs. The proposed processor has been embedded in an SoC for video processing and proven in silicon.

This paper is organized as follows. In Section 2, we present the overview of the EISC architecture. Section 3 describes the proposed schemes to enhance the performance of DSP applications. In Section 4, we present the microarchitecture of the DSP-enhanced processor that we propose and the implementation issues. The performance evaluation of the proposed microprocessor is shown in Section 5. Then, we conclude our paper in Section 6.

2 The EISC Architecture

Code density, chip area, and power consumption are three major design issues of the embedded microprocessor. These three features are closely related to each other, so none of these can be overlooked. However, many 32-bit embedded microprocessors suffer from poor code density. In order to address this problem,

some RISC-based 32-bit microprocessors adopt 16-bit compressed instruction set architectures, such as ARM THUMB [7] and MIPS16 [8]. These approaches provides better code density but need some mechanisms to extend the insufficient immediate field and to provide backward compatibility with their previous architectures, which can result in extra hardware overheads. Moreover, these architectures have difficulty in utilizing their registers efficiently in the compressed mode. [7,8].

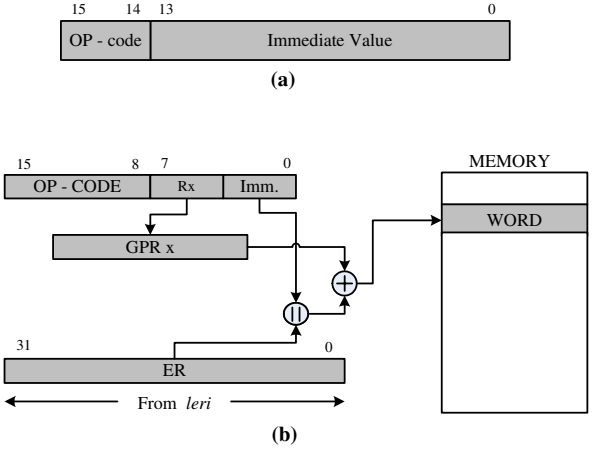


Fig. 1. Use of the `leri` instructions. (a) Format of the `leri` instruction. (b) Preloaded immediate value in ER combined with a short immediate value in an instruction.

The EISC architecture takes a different approach for achieving high code density and a low memory access rate [3,4]. The EISC uses an efficient fixed-length 16-bit instruction set for 32-bit data processing. To resolve the problem of insufficient immediate operand fields in a terse way, EISC uses an independent instruction called `leri`, which consists of a 2-bit opcode and a 14-bit immediate value as shown in Fig. 1(a). The `leri` instruction can extend the immediate field by loading immediate value to a special register called ER (extension register.) Several `leri` instructions can be combined with one related instruction to form a long immediate operand. A similar approach was used in a 16-bit instruction set proposed in [9], in which a long immediate operand was prepared in a register by loading the value from memory or by computing with a sequence of instructions such as a move-shift-add sequence. The EISC uses more efficient and more flexible approach. Fig. 1(b) shows an example in which the preloaded immediate value in ER can be combined with a short immediate value in an instruction.

By using the `leri` instruction, the EISC architecture can make the program code more compact than the competing architectures, ARM-THUMB and MIPS16, since the frequency of the `leri` instruction is much less than 20% in most programs. In [3], the code density of the EISC architecture was evaluated to be 6.5% higher than that of ARM THUMB and 11.5% higher than that of

MIPS16, for various programs considered in [3]. Moreover, the overhead from the additional `leri` instructions can be minimized by the `leri` instruction folding method explained in [11].

As the EISC uses the `leri` instructions, it can keep its instructions sized equal, and make its instrcution decoder much simpler than that of the CISC (Complex Instruction Set Computer) arcthiecture. In addition, the EISC does not suffer from the overheads for switching its processor mode, which is often needed to handle long immediate values or complicated instructions in the compressed code RISC architectures. In addition, the EISC architecture reduces its data memory access rate by fully utilizing its 16 registers while the competing architectures can access limited number of registers in the compressed mode. The data memory access rate of the EISC is 35.1% less than that of the ARM THUMB and 37.6% less than that of MIPS16 by the benchmarks in [4]. Thus, the EISC can reduce both instruction references and data references. Reducing the memory accesses would bring reduction in power consumption related to the memory accesses and also lessen the performance degradation caused by the speed gap between the processor and the memory. Moreover, the EISC-based system can be implemented in a smaller die, because the memory circuit can be made smaller.

It is another advantage of using the `leri` instruction that the EISC programmers normally do not have to distinguish the instructions for the register-immediate operation from those for the register-to-register operation. They are distinguished by the existence of the preceding `leri` instruction. Most instruc-

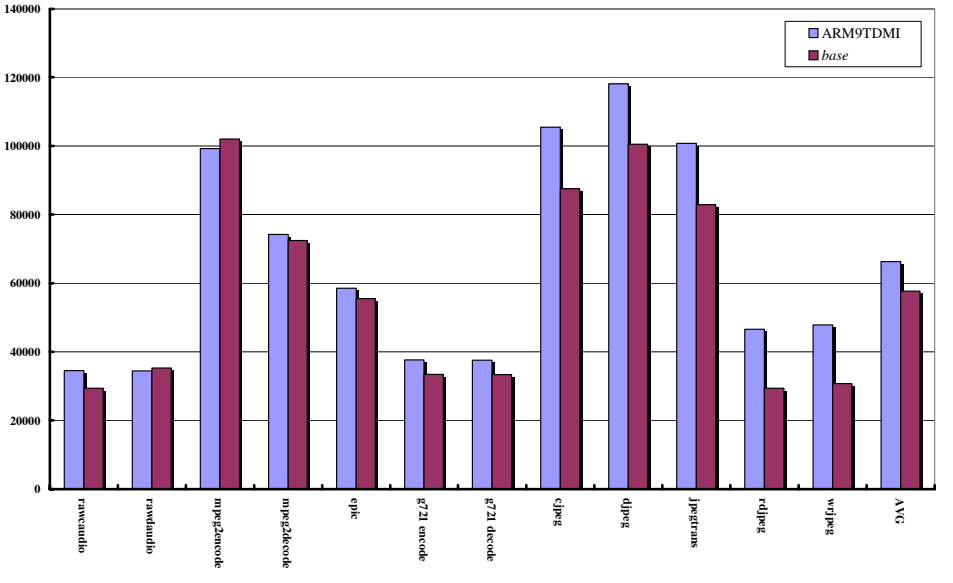


Fig. 2. Static code sizes of the programs in Mediabench. GCC 3.2 was used with the same optimization options.

tions for arithmetic operations in the EISC are of register-to-register type. When more than one `leri` instruction precedes an arithmetic instruction, the arithmetic instruction is identified as a register-immediate-type one. This feature helps save the code space more.

We compare the static code sizes of the programs in Mediabench [6] compiled on *base* and ARM9-TDMI by GCC 3.2 with the same optimization options to show the code density of the EISC in the DSP applications. The results are summerized on the Fig. 2. On average, *base* has about 18.9% higher code density than ARM9-TDMI for the benchmarks considered in this paper.

3 DSP Enhancements for the EISC Processor

The processor *base* is a 5-stage pipelined implementation of the 32-bit EISC architecture and adopts a microarchitecture that is similar to the one in [5]. It is equipped with simple DSP capabilities, such as a single-cycle 32b×32b=64b signed/unsigned multiplication, a single-cycle 32b×32b+64b=64b MAC (multiplyaccumulate) operation, a 32-bit (barrel) shift operation, and a leading one or zero count operation that can be used in Huffman decoding [12].

In this paper, we propose several DSP enhancements for developing a DSP-enhanced EISC processor with as little extra hardware cost as possible. We develop instructions for supporting DSP: instructions for SIMD operations with the capability of saturation arithmetic; and instructions for generating addresses and packing, loading, and storing media data. We try to minimize the overheads for feeding data into the SIMD unit [13]. We develop the enhancements so that they can be realized within the limited code space, since the processor uses 16-bit instructions (even though it has a 32-bit datapath.)

3.1 DSP Instruction Set

Since the data in the multimedia applications are frequently smaller than a word, we can boost up the performance of the processor base by adopting a SIMD architecture. When the SIMD operations are performed, however, some expensive packing operations are also performed for arranging the data to the SIMD operation unit [13]. In order to reduce the number of packing operations, the processor should support various types of packed data. However, since the code space allowed for the DSP-enhancements is limited as mentioned above, we focus on accelerating only the MAC operations which are apparently the most popular operations in the DSP applications. We implement two types of MAC operations shown in Fig. 3: the parallel SIMD type and the sum-of-products type. The parallel SIMD type of MAC operation, shown in Fig. 3(a), is much more efficient than any other types of operations, for processing the arrays of structured data such as RGBA formatted pixel data. On the other hand, the sum-of-products type of MAC operation, shown in Fig. 3(b), is efficient for general data arrays.

We also adopt the instructions for the saturation arithmetic which is often used in DSP applications. Unlike the wrap-around arithmetic, the saturation

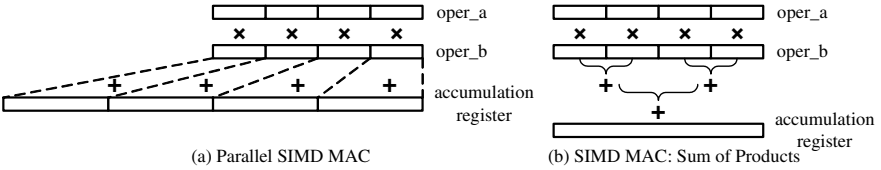


Fig. 3. Two types of the SIMD MAC operations supported in the proposed DSP-enhanced processor

arithmetic uses the maximum or the minimum value when it detects an overflow or an underflow.

In the signal processing, the fixed-point arithmetic is commonly used because it is much cheaper and faster than the floating-point arithmetic. During the multiplication of two 32-bit fixed-point numbers, the results are stored in the 64-bit multiplication result register (MR). Since the position of radix point in a fixed-point number can be changed during multiplications, we need to select a part of the 64-bit multiplication result to make a 32-bit fixed-point number. For that purpose, *base* would require a sequence of five instructions looking like the one in the first box shown below. We propose to use an instruction with the mnemonic `mrs` (multiplication result selection), as shown below.

```
mfmh    %Rx             # move MR(H) to GPR(Rx)
mfml    %Ry             # move MR(L) to GPR(Ry)
asl     DP, %Rx         # DP-bit shift left Rx
lsr     (32-DP), %Ry    # (32-DP)-bit shift right Ry
or      %Rx, %Ry        # Ry = Rx | Ry
```

↓

```
mrs     DP, %Ry
```

3.2 Accelerating Address Generation

DSP applications usually perform a set of computations repeatedly for streaming data that have almost uniform data patterns. In this section, we introduce a loop-efficient hardware address generator that can accelerate the memory-addressing processes. We intend to use this address generator for accelerating DSP applications, including the memory-copy operations, with low cost.

The proposed address generator has a capability of handling the auto-increment addressing mode which is usually used for various applications including the memory-copy operations. The address generator is also intended for supporting other special memory-addressing modes, such as the wrap-around incremental addressing mode and the bit-reversal addressing mode which are commonly used in many DSP applications. We only support the post-modifying scheme because it is more popularly used [14].

Since the hardware complexity is important, the proposed address generator only produces the offsets instead of the entire addresses. Fig. 4 shows the block

diagram of the proposed address generator. For the post-increment addressing scheme, the generator uses a counter to produce the desired offset and an adder to build the wanted address. The generated offset is written back to the counter register. The control register is used to control the address generator and holds a control word regarding generation pattern, offset counter, end offset for looping, and incremental offset. Depending on the operation mode, the control word is configured in one of three formats, one of which is shown in Fig. 5.

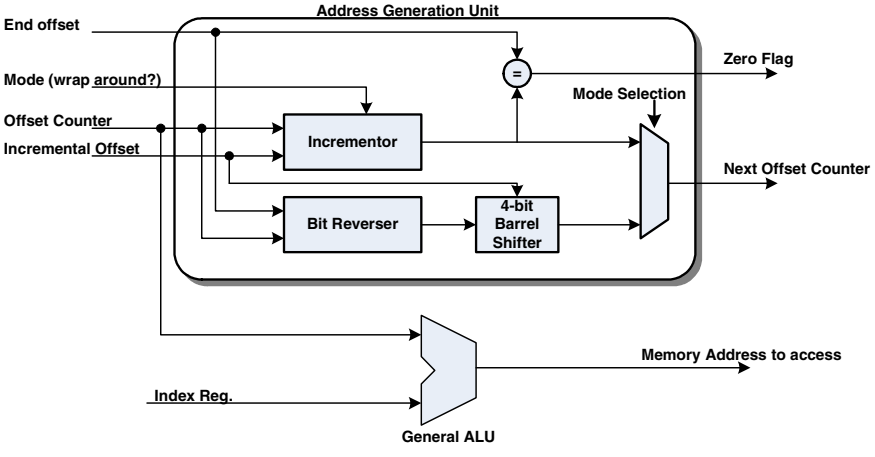


Fig. 4. Proposed address generation unit

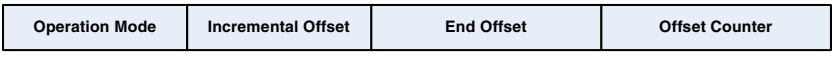


Fig. 5. One of three formats of the control word for the proposed address generation unit. The format of the registers depends on the operation mode.

DSP applications often use a specific computation kernel repeatedly, which is usually implemented as a loop in the programs. For example, a load-computation-store-test-branch sequence represents a computation for streaming data. The proposed address generator is equipped with a comparator to detect the end offset. When the address reaches to the final address, the address generator sets a flag related to the appropriate conditional branch.

3.3 Register Extension

Some popular DSP routines, especially those routines for coding transforms, use many temporary values simultaneously. However, most embedded microprocessors do not have enough registers to hold those temporary values. A general

approach for resolving this problem of shortage of registers is to spill the contents of registers into the memory. However, media data often take the form of data streams, in which case they are temporarily stored in the memory, continuously read into the processor, and used only a few times. Therefore, media applications running on an embedded processor often suffer from significant performance losses due to the frequent register spilling processes.

We propose a register extension scheme that increases the effective number of registers by adopting the idea of shadow register to hold temporary values in the registers. The idea of shadow register has been used in various aspects, such as the context switching latency [15]. We find that this idea is also very useful for those processors which have limited code spaces, such as the EISC processors, since it is hard for them to allocate additional bits for register indexing [10].

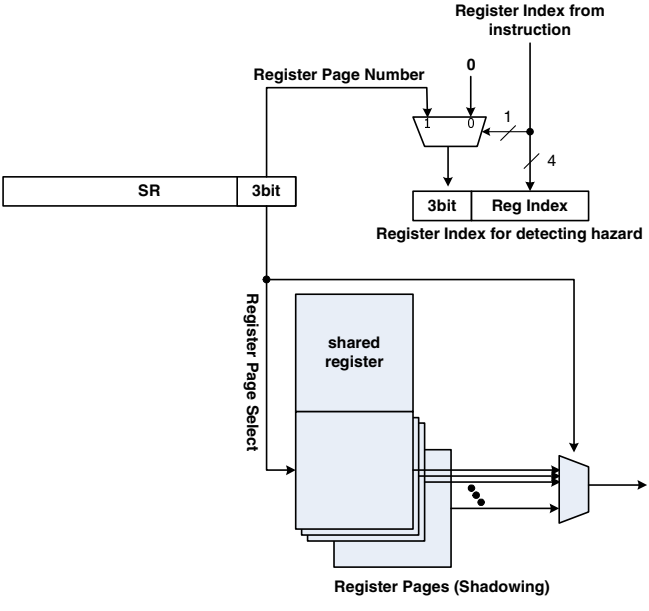


Fig. 6. The register extension scheme. The active register page is selected by the designated three bits in the status register (SR).

Fig. 6 shows the proposed register extension scheme. The register file consists of a set of smaller register files, named *register pages*. We divide the register file into two parts: one part consists of the registers to be shared among pages; and another part consists of the registers to be shadowed. The use of shared registers reduces the need of the instructions added to support data movement among register pages. In order to select the active register page, we allocate three bits in the status register. Thus, we can use up to eight register pages. When the processor need to check data dependencies, it constructs a new register index by merging the page selection bits with the register number defined in the instruction, as shown in Fig. 6.

3.4 Power Management

It is important to support a power management scheme for reducing the power consumption in an SoC. The EISC processor offers the `halt` instruction for switching the processor and the peripheral devices to the power-down mode [12]. The proposed processor defines a dedicated power management protocol for the `halt` instruction. The `halt` instruction can have immediate values between 0 and 15. For the value of 0 through 3, the processor backs up the current context and switches the state of the processor to the HALT state that is a safe state of the power-down mode. During the process of switching to the HALT state, the processor sends a command to the power management unit in the SoC using a dedicated protocol.

The power management unit controls the power state (clock frequency) of each device. To wake up the processor, it needs an external interrupt or an NMI(non-maskable interrupt) which makes the power management unit to regain the power state for the processor. After that, the processor switches its state to the wake-up state and processes the interrupt. Fig. 7 shows the protocol of the power management unit for the SoC that uses the proposed processor.

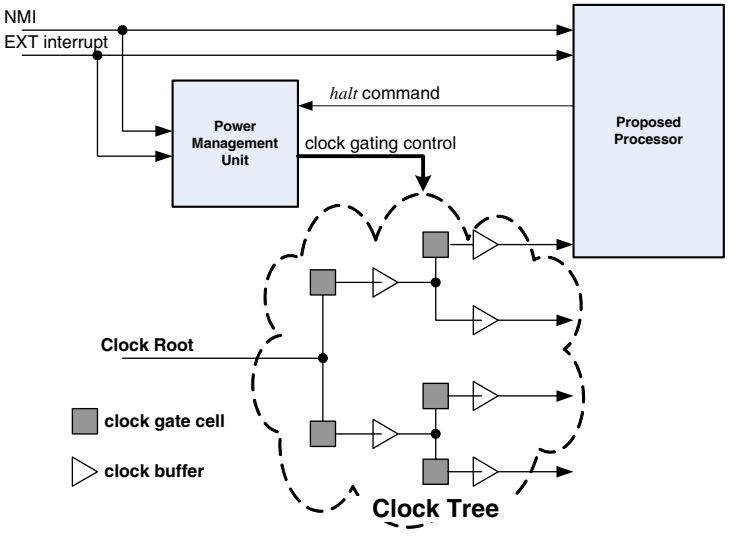


Fig. 7. Power management

In order to reduce the power consumption further, we also apply the clock-gating scheme to the pipeline flip-flops and the register file. We control the clock for the pipeline flip-flops using the pipeline control signals. In addition, we also apply the operator isolation scheme to the computation units in the EX stage to reduce the power consumption in the unused arithmetic units.

4 Implementation

We have designed the proposed DSP-enhanced EISC processor. Fig. 8 shows the microarchitecure of our design. Fig. 8 also shows how the proposed processor can be integrated with other coprocessors (outlined by the dotted line.) The shaded boxes represent the blocks modified or added to enhance the DSP capabilities. In Fig. 8, the block MBMU (Memory Bank Management Unit) provides the information for configuring memory, such as signals regarding access right, cache enable/disable, and TLB enable/disable. The block OSI (On-Silicon ICE breaker) provides an on-chip debugging capability such as breakpoint and watch point.

As shown in Fig. 8, a SIMD MAC (multiply-accumulate) unit is located at EX and MEM stage. We designed the architecture for the SIMD MAC unit so that it can be included in the existing processor (*base*) with minimal impact on clock frequency, pipelines, and area. To minimize area overhead, we merge SIMD MAC unit and existing MAC unit. In *base*, the 32b $\times$ 32b = 32b multiplication is frequently used to generate the address for designating an element in an array or a structures. Therefore, if an additional clock cycle is added for this operation, it introduces a bubble in the pipeline due to the data dependency. However, the critical-path delay of a single-cycle SIMD MAC unit would be too large for our processor. To resolve this problem, we design a two-stage pipelined SIMD MAC unit.

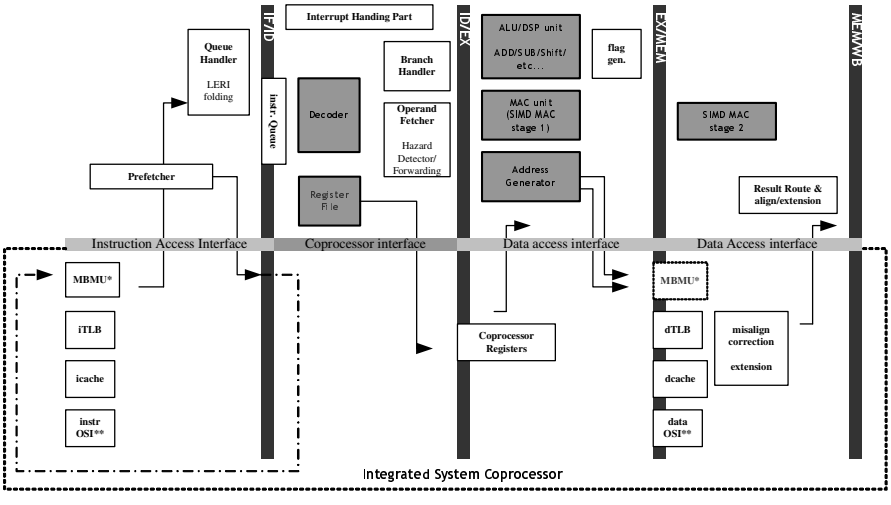


Fig. 8. The microarchitecture of the proposed DSP-enhanced microprocessor. The figure also shows how the proposed processor can be integrated with other coprocessors (outlined by the dotted line.) Shadowed boxes represent the added and the modified blocks to accelerate DSP applications. The block MBMU configures memory, while the block OSI provides on-chip debugging capabilities.

The address generation unit is located in the EX stage. The generated offset is written back to the control register that is located at the ID/EX pipeline flip-flop. In the ID state, we implement the register file of two register pages. Thus, we have twenty-four GPRs in the processor.

5 Evaluations

The designed DSP-enhanced microprocessor has been modeled in Verilog HDL as a synthesizable RTL model. We synthesize the model using the Samsung STD130 0.18μm CMOS standard cell library [16] and the Synopsys DesignCompiler. The critical path delay is estimated by using the Synopsys PrimeTime under the worst-case operating condition (1.65V supply voltage and 125 °c junction temperature.) The results are summarized in Table 1. As shown in Table 1, the critical path delay is almost the same even through some units are added to accelerate DSP applications. The enhancements we add cost about 10270 equivalent gates, most of which are used for the shadow register file and the SIMD MAC unit. The others cost less than 1000 gates per each unit. For example, the address generation unit costs 917 equivalent gates.

Table 1 also shows the performance of the dynamic power management scheme we has implemented. As shown in Table 1, we observe about 65.5% reduction in the power consumption by clock gating. Note that the area occupied by the processor becomes smaller, while the critical path delay remains almost the same when we implement the dynamic power management scheme. It is because we can eliminate those multiplexers for controlling the pipeline by using the gates for gating clocks.

Table 1. Cost and power performance of the enhancements

	Area [equi. gates]	Critial Path Delay [ns]	Power Consumption [mW]
Base processor	56852	6.25	36.33
Proposed processor	67122	6.19	42.47
Proposed processor(lp)	55191	6.21	14.44

In order to evaluate the performance of the proposed architecture and compare it with that of the ARM9EJ-S, we used DSP kernels: IMDCT (inverse modified discrete cosine transform), FIR (Finite Impulse Response) filter and ADPCM (Adaptive Differential Pulse Code Modulation) encoder and decoder. The IMDCT routine exploit the 32-bit operations for high fidelity audio coding applications including MP3, and AAC. The FIR filter is the most common DSP function in many DSP applications. The ADPCM is commonly used in storing sound files. We compare the code sizes and the peformances of the proposed processor with those of ARM9EJ-S using the original C programs and the DSP-optimized assembly programs for each processors considered. When we optimize the DSP-kernel, we don't use SIMD MAC instructions for fair comparison.

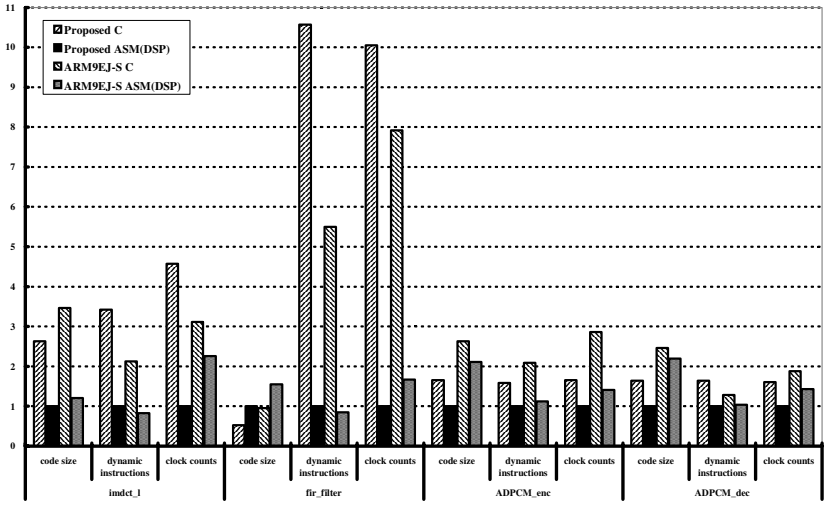


Fig. 9. Comparison of the code sizes and the performances of the prospoed processor and those of ARM9EJ-S. The values are normalized with respect to the DSP-optimized program on the proposed processor.

The results of the experiments are summarized in Fig. 9. The proposed DSP architecture reduces the average execution time of the DSP kernels considered by 77.6%, and it outperforms the DSP-optimimzed ARM9EJ-S by 40.8%. Meanwhile, the code size of the propsoed processor is about 43.3% smaller than that of the ARM9EJ-S. Peformance advantage is mainly due to the use of single cycle $32 \times 32 + 80 = 80$ MAC operation and the use of LERI folding method [11] that handles `leri` instructions independently.

We also experiment with real DSP applications. We use the MAD (MPEG audio decoding) MP3 decoding library in the MediaBench [6] and a simple MP3 player program, *minimad*, that uses the MAD library. In our experiment, we use a sample 8.124-second stereo MP3 clip that is encoded at 44.1KHz sampling rate and 192bps bit rate. As the programs for testing our DSP-enhancement schemes, we select and optimize two DSP kernels, *imdct_l* and *subband synthesis*. Our experiment shows that about 47.7% of the run time of the MP3 decoding is spent on running these selected kernels. Another reason why we select these kernels is that they use macro functions coded in the assembly language. Thus we are able to exclude the effects that would be caused by the compiling process for compiling the kernels coded in high-level languages.

We measure the clock counts. As shown in Fig. 10, the optimized *imdct_l* kernel reduces the run time of the MP3 decoder by 16.8%, while the optimized *subband synthesis* kernel reduces it by 14.0%. When we optimize both *imdct_l* and *subband synthesis* kernels, we observe about 30.9% reduction in the run time of the MP3 decoder. As a result, the proposed DSP-enhanced processor is able to decode a high-fidelity MP3 audio at 31.1MHz, while the processor *base* has

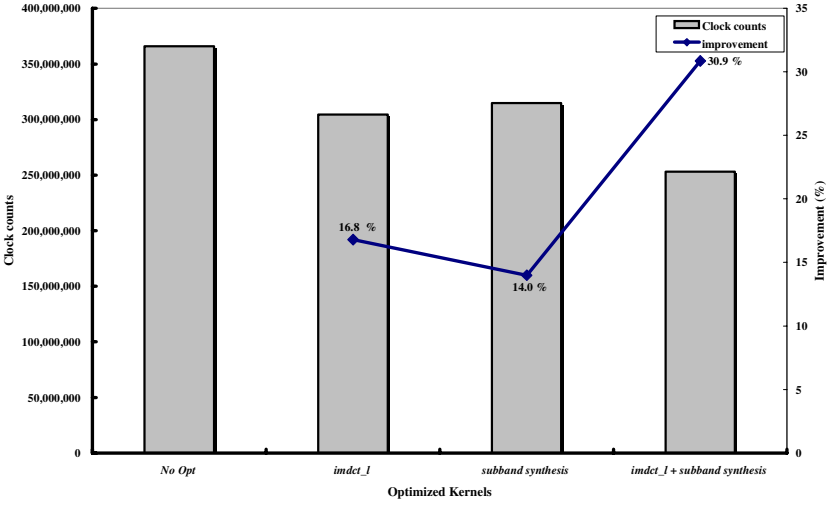


Fig. 10. Clock counts of the MP3 decoding program when the selected kernel(s) is(are) optimized

to be clocked at 45MHz to perform the same job. The proposed DSP-enhanced processor has been embedded in an SoC for video processing [17] and proven in silicon.

6 Conclusions

In this paper, we have introduced a DSP-enhanced embedded microprocessor based on the EISC architecture. In order to accelerate DSP application with as little extra hardware as possible, we propose various enhancement schemes: some schemes exploit the special features of the EISC, including the `leri` instruction; and some schemes are to overcome the inherent deficits of the EISC, including the insufficiency of the instruction bits and the insufficiency of GPRs.

We adopt the SIMD architecture and tailor it to reduce the hardware complexity and the packing overhead. To improve the performance of SIMD architecture, we propose a loop-efficient address generation unit. The proposed address generation unit is designed to support commonly used memory addressing modes in DSP applications with low hardware complexity. We also adopt a register extension scheme to reduce performance degradation due to the register spilling.

The proposed DSP-enhanced processor has been modeled in Verilog HDL and synthesized using a 0.18μm CMOS standard library. The proposed DSP enhancements cost about 10300 gates and not increase the clock frequency. Our simulations and experiments show that the proposed DSP-enhanced processor reduces the execution time of the DSP kernels considered in this work by 77.6% and the MP3 applications by 30.9%.

The high code density of EISC is an inherent advantage to a low-power embedded system. In addition, we implement the supports for static and dynamic power managements. We exploit the supports for power management and reduce the power consumption by 65.5%, while our DSP enhancements increase the power consumption by 16.9%. We also introduce our power management scheme for SoC. The proposed processor has been embedded in an SoC for video processing and proven in silicon.

Acknowledgements

The authors wish to acknowledge the CAD tool support of IDEC (IC Design Education Center), Korea and the financial support of Advanced Digital Chips Inc., Korea. The authors would also like to thank the anonymous reviewers for their valuable comments.

References

1. Francis, H.: ARM DSP-Enhanced Instructions White Paper, http://arm.com/pdfs/ARM-DSP.pdf
2. MIPS Tech. Inc.: Architecture Set Extension, http://www.mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/doclibrary
3. Cho, K.Y.: A Study on Extendable Instruction Set Computer 32 bit Microprocessor, J. Inst. of Electronics Engineers of Korea, **36-D(55)** (1999) 11–20
4. Lee, H., Beckett, P., Appelbe, B.: High-Performance Extendable Instruction Set Computing, Proc. of 6th ACSAC-2001 (2001) 89–94
5. Kim, H.-G., Jung, D.-Y., Jung, H.-S., Choi, Y.-M., Han, J.-S., Min, B.-G., Oh, H.-C.: AE32000B: A Fully Synthesizable 32-bit Embedded Microprocessor Core, ETRI Journal **25(5)** (2003) 337–344
6. Lee, C., Potkonjak, M., Mangione-Smith. H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, MICRO-30(1997) 330–335
7. ARM Ltd.: The Thumb Architecture Extension, http://www.arm.com/products/CPUs/archi-thumb.html
8. Kissell, K.D.: MIPS16: High-density MIPS for the Embedded Market, MIPS Tech. Inc., http://www.mips.com/Documentation/MIPS16whitepaper.pdf
9. Bunda,J.D.: Instruction-Processing Optimization Techniques for VLSI Microprocessors, PhD thesis, The University of Texas at Austin (1993)
10. Park, G.-C., Ahn, S.-S., Kim, H.-G., Oh, H.-C.: Supports for Processing Media Data in Embedded Processors, Poster Presentation, HiPC2004 (2004).
11. Cho,K.Y., Lim, J.Y., Lee, G.T., Oh, H.-C., Kim, H.-G., Min, B.G., Lee, H.: Extended Instruction Word Folding Apparatus, U.S. Patent No.6,631,459, (2003)
12. Kim, H.-G.: AE32000: Hardware User Guide, http://adc.co.kr/Korean/Products/ProdDocu/ae32000b_hw_ug_v1.1_031118e.pdf
13. Talla, D., John, L.K., Buger, D.: Bottlenecks in Multimedia Processing with SIMD Style Extensions and Architectural Enhancements, IEEE Tras. of Comp. **52(8)** (2003) 1015–1011
14. Hennessy, J.L, Patterson, D.A.: Computer Architecture; A Quantitative Approach 3rd Ed., Morgan Kaufmann Publishers (2003)

15. Jayaraj, J., Rajendran, P.L., Thirumoolam, T.: Shadow Register File Architecture: A Mechanism to Reduce Context Switch Latency, HPCA-8 (2002) Poster Presentation
16. Samsung Electronics: STD130 0.18um 1.8V CMOS Standard Cell Library for Pure Logic Products Data Book, Samsung Electronics (2001)
17. Advanced Digital Chips Inc.: GMX1000: A High Performance Multimedia Processor User Manual, Advanced Digital Chips Inc. (2005)

Author Index