André Seznec Joel Emer
Michael O'Boyle Margaret Martonosi
Theo Ungerer (Eds.)

LNCS 5409

High Performance Embedded Architectures and Compilers

Fourth International Conference, HiPEAC 2009
Paphos, Cyprus, January 2009
Proceedings

Springer

Lecture Notes in Computer Science 5409

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

André Seznec Joel Emer
Michael O'Boyle Margaret Martonosi
Theo Ungerer (Eds.)

High Performance Embedded Architectures and Compilers

Fourth International Conference, HiPEAC 2009
Paphos, Cyprus, January 25-28, 2009
Proceedings

Springer

Volume Editors

André Seznec
IRISA, Campus de Beaulieu
35042 Rennes Cedex, France
E-mail: seznec@irisa.fr

Joel Emer
Intel Corporation, Massachusetts Microprocessor Design Center
Hudson MA 01749, USA
E-mail: joel.emer@intel.com

Michael O'Boyle
School of Informatics, Institute for Computing Systems Architecture
Edinburgh EH9 3JZ, United Kingdom
E-mail: mob@inf.ed.ac.uk

Margaret Martonosi
Princeton University, Department of Electrical Engineering
Princeton, NJ 08544-5263, USA
E-mail: mrm@ee.princeton.edu

Theo Ungerer
University of Augsburg, Department of Computer Science
86135 Augsburg, Germany
E-mail: ungerer@informatik.uni-augsburg.de

Library of Congress Control Number: 2008942486

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

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

ISSN 0302-9743
ISBN-10 3-540-92989-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-92989-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12601547 06/3180 5 4 3 2 1 0

Preface

HiPEAC 2009 was the fourth edition of the HiPEAC conference series. This conference series is largely associated with the FP7 Network of Excellence HiPEAC2. The first three editions of the conference in Barcelona (2005), Ghent (2007) and Göteborg (2008) attracted a lot of interest with more than 200 attendees at the last two editions and satellite events. It is a great privilege for us to welcome you to the fourth HiPEAC conference in the beautiful, touristic city of Paphos, Cyprus.

The offerings of this conference are rich and diverse. We offer attendees a set of seven workshops on topics that are central to the HiPEAC network roadmap: multi-cores, simulation and performance evaluation, compiler optimizations, design reliability, reconfigurable computing, and interconnection networks. Additionally, a tutorial on design reliability is offered.

The conference program was as rich as last year's. It featured many important and timely topics such as multi-core processors, reconfigurable systems, compiler optimization, power-aware techniques and more. The conference also offered two keynote speeches: Tilak Agerwala from IBM Research presenting the view from a major industry player, and François Bodin from CAPS-Entreprise presenting the view of a start-up.

There were several social activities during the conference offering ample opportunity for informal interaction. These included a reception, an excursion to various archeological sites and a banquet at a traditional tavern.

This year we received 97 paper submissions, of which 14 were co-authored by a Program Committee member. Papers were submitted from 20 different nations (approximately 46% from Europe, 15% from Asia, 32% from North America, 4% from Africa and the Middle East, and 3% from South America), which is an indicator of the global visibility of the conference.

We had the luxury of having a strong Program Committee consisting of 30 experts in all areas within the scope of the conference. Each paper was typically reviewed by four Program Committee members and in those cases where there was a divergence of views, additional external reviews were sought. In all, we collected a total of 394 reviews and we were happy to note that each paper was rigorously reviewed before we made our final decision.

The Program Committee meeting was held in the new Informatics Forum building at the University of Edinburgh, UK. Despite a long trip for many members of the Program Committee, 20 attended the meeting. For virtually all papers, at least two reviewers were present in person. Other members participated actively by telephone. The Program Committee meeting was also preceded by an e-mail discussion of papers among the reviewers.

At the Program Committee meeting the non-Program Committee-authored papers were discussed in the order of average score. When a paper was discussed

where a participating member was either a co-author or had conflicts with that paper, that person left the room. Program Committee-authored papers were discussed in a separate block of time, and order was randomized to give Program Committee members minimal information about the ranking of their paper. At the end, we accepted 27 papers of which 8 are Program Committee papers, yielding an acceptance rate of 28%.

The end result of the whole effort was a high-quality and interesting program for the HiPEAC 2009 event.

The planning of a conference starts well in advance. Were it not for the unselfish and hard work of a large number of devoted individuals, this conference would not have been as successful as it was. First of all, a special thanks goes to Yanos Sazeides from the University of Cyprus, who took care of the local organization and more. Without his dedication and hard work, this conference could never have been organized. We would like to thank all the team that worked to make this conference successful: Stefanos Kaxiras (Patras) for putting together an attractive pre-conference program; Wouter De Raeve (Ghent), our Finance Chair for running the books; Hans Vandierendonck (Ghent) for timely publicity campaigns; Basher Shehan and Ralf Jahr (Augsburg) for the hard work in putting together the proceedings; Michiel Ronsse (Ghent) for administering the submission and review system; and finally Sylvie Detournay and Klaas Millet (Ghent) for administering the Web. We would also like to thank Per Stenström, Steering Committee Chair, and Koen De Bosschere, Chair of the HiPEAC2 network, for the advice they provided with us.

Thanks to all of you!

Finally, we would also like to mention the support from the Seventh Framework Programme of the European Union, represented by project officer Panagiotis Tsarchopoulos, for sponsoring the event and for the travel grants.

October 2008

André Seznec
Joel Emer
Mike O'Boyle
Margaret Martonosi

Organization

Executive Committee

General Co-chairs	André Seznec IRISA/INRIA, France Joel Emer Intel/MIT, USA
Program Committee Co-chairs	Michael O'Boyle University of Edinburgh, UK Margaret Martonosi Princeton University, USA
Workshop/Tutorials Chair	Stefanos Kaxiras University of Patras, Greece
Local Arrangements Chair	Yiannakis Sazeides University of Cyprus, Cyprus
Finance Chair	Wouter De Raeve Ghent University, Belgium
Publicity Chair	Hans Vandierendonk Ghent University, Belgium
Publications Chair	Theo Ungerer University of Augsburg, Germany
Submissions Chair	Michiel Ronsse Ghent University, Belgium
Web Chair	Klaas Millet Ghent University, Belgium

Program Committee

David Albonesi	Cornell University, USA
Eduard Ayguade	UPC, Spain
François Bodin	CAPS Entreprise, France
John Cavazos	University of Delaware, USA
Albert Cohen	INRIA, France
Marco Cornero	STmicroelectonics, Italy
Lieven Eeckhout	Ghent University, Belgium
Guang Gao	University of Delaware, USA
Thomas Gross	ETH Zurich, Switzerland
Michael Hind	IBM, USA
Mary J. Irwin	Pennsylvania State University, USA
Russell Joseph	Northwestern University, USA
Ben Juurlink	TU Delft, The Netherlands
Stefanos Kaxiras	University of Patras, Greece

Rainer Leupers	University of Aachen, Germany
Gabriel Loh	Georgia Tech University, USA
José Martinez	Cornell University, USA
Bilha Mendelson	IBM Research, Israel
Andreas Moshovos	University of Toronto, Canada
David Padua	UIUC, USA
Markus Pueschel	CMU, USA
Li-Shiuan Peh	Princeton University, USA
Lawrence Rauchwerger	Texas A&M University, USA
Ronny Ronen	Intel Corp., Israel
Yiannakis Sazeides	University of Cyprus, Cyprus
Vassos Soteriou	Cyprus University of Technology, Cyprus
Per Stenström	Chalmers University, Sweden
Olivier Temam	INRIA, France
M. J. Thazhuthaveetil	Indian Institute of Science, India
Richard Vuduc	Georgia Tech University, USA

Steering Committee

Per Stenström	Chalmers University of Technology, Sweden
Anant Agarwal	MIT, USA
Koen De Bosschere	Ghent University, Belgium
Michel Dubois	University of Southern California, USA
Rajiv Gupta	University of California, Riverside, USA
Wen-mei W. Hwu	UIUC, USA
Manolis Katevenis	University of Crete/Forth, Greece
Theo Ungerer	University of Augsburg, Germany
Mateo Valero	UPC/BSC, Spain

Additional Reviewers

Virat Agarwal	Saugata Ghose	Joseph Manzano
Erik Altman	Boris Ginsburg	Avi Mendelson
Matthew Arnold	R. Govindarajan	Maged Michael
Michael Behar	Daniel Gracia-Perez	Pierre Michaud
Vicenc Beltran	Zvika Guz	Peter Milder
Mauro Bianco	Sunpyo Hong	Gilles Mouchard
Evgeny Bolotin	Engin Ipek	Janani Mukundan
Ioana Burcea	Hyesoon Kim	Onur Mutlu
Nathan Clark	Nevin Kirman	Rishiyur Nikhil
Mattan Erez	Meyrem Kirman	Daniel Orozco
Dongrui Fan	Marios Kleanthous	Ioannis Papadopoulos
Franz Franchetti	Jakub Kurzak	Maikel Pennings
Ron Gabor	Xiaoming Li	Jason Riedy

Erven Rohou
Hongbo Rong
Roni Rosner
Efraim Rotem
Efi Rotem
Simon Rubanovich
Gad Sheaffer
Tim Smith
Jared Stark
Arthur Stoutchinin
Gabriel Tanase
Nathan Thomas
Xinmin Tian
Osman Unsal
Chronis Xekalakis
Handong Ye
Jason Zebchuk

Table of Contents

Invited Program

I Dynamic Translation and Optimisation

II Low Level Scheduling

III Parallelism and Resource Control

Keynote: Challenges on the Road to Exascale Computing

Tilak Agerwala

VP of Systems, IBM Research

Abstract. Supercomputing systems have made great strides in recent years as the extensive computing needs of cutting-edge engineering work and scientific discovery have driven the development of more powerful systems. In 2008, we saw the arrival of the first petaflop machine, which quickly topped the Top500 list, while also occupying the number one position on the Green500 list. Historic trends indicate that in ten years, we should be at the exascale level. We believe that applications in many industries will be materially transformed by exascale systems and will drive systems not just to 1000X in raw performance but to equally dramatic improvements in data intensive computing and real time stream processing.

Meeting the exascale challenge will require significant innovation in technology, architecture and programmability. Power is a fundamental problem at all levels; traditional memory cost and performance are not keeping pace with compute potential; the storage hierarchy will have to be re-architected; networks will be a much bigger part of the system cost; reliability at exascale levels will require a holistic approach to architecture design, and programmability and ease-of-use will be an essential component to extract the promised performance at the exascale level.

In this talk, I will discuss the major challenges of exascale computing, touching on the areas of technology, architecture, reliability, programmability and usability.

Biography of Tilak Agerwala

Tilak Agerwala is vice president, Systems at IBM Research. He is responsible for developing the next-generation technologies for IBM's systems, from microprocessors to commercial systems and supercomputers, as well as novel supercomputing algorithms and applications. Dr. Agerwala joined IBM at the T.J. Watson Research Center and has held executive positions at IBM in research, advanced development, marketing and business development. His research interests are in the area of high performance computer architectures and systems. Dr. Agerwala received the W. Wallace McDowell Award from the IEEE in 1998 for outstanding contributions to the development of high performance computers. He is a founding member of the IBM Academy of Technology. He is a Fellow of the Institute of Electrical and Electronics Engineers. He received his B.Tech. in electrical engineering from the Indian Institute of Technology, Kanpur, India and his Ph.D. in electrical engineering from the Johns Hopkins University, Baltimore, Maryland.

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, p. 1, 2009.

Keynote: Compilers in the Manycore Era

François Bodin

CTO, CAPS

Abstract. Homogeneous multicore processors such as the ones proposed by Intel or AMD have become mainstream. However heterogeneous architectures, such as multicore associated with GPUs or with any hardware specialized co-processors usually offer a much higher peak performance/power ratio. When high performance and power efficiency has to be achieved, specialized hardware is often the way to go. Combining a general-purpose multicore with a highly parallel coprocessor, e.g., GPU, allows to both achieve high speedups on parallel sections while maintaining high performance on control sections.

However, programming such heterogeneous architectures is quite a challenge for any application developer. The embedded market has been living with it for decades but at a very high programming cost. The general-purpose computing is now entering this era.

Manycore performance opens HPC to many new scientific and consumer applications. New multimedia, medical and scientific applications will be developed by hundreds of thousands of engineers across the world. These applications, usually provided by ISV, will have to be tuned for thousands of various platform configurations built with different hardware units such as CPUs, GPUs, accelerators, PCIx buses, memories, etc., each configuration having its own performance profile. Furthermore, in most manycore systems, applications are in competition at run-time for hardware resources like the memory space of accelerators. If ignored this can lead to a global performance slowdown.

The past of parallel programming is scattered with hundreds of parallel languages, most of them were designed to address homogeneous architectures and concerned a small and well-trained community of HPC programmers.

With the new diversity of parallel hardware platforms and the new community of non-expert developers, expressing parallelism is not sufficient anymore. Resources management, application deployment, and portable performance are interconnected issues that require to be addressed holistically.

As many decisions should be taken according to the available hardware, resources management cannot be moved apart from parallel programming. Deploying applications on various systems without having to deal with thousands of configurations is the main concern of ISVs. The grail of parallel computing is to be able to provide portable performance by making hardware changes transparent for a large set of machines and fluctuating execution contexts.

Are general-purpose compilers obsolete?

Compilers are keystone solutions of any approaches that deal with previous challenges. But general-purpose compilers try to embrace so

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 2–3, 2009.

many domains and try to serve so many constraints that they frequently fail to achieve very high performance. They need to be deeply revisited!

Recent techniques are showing promises. Iterative compilation techniques, exploiting the large CPU cycle count now available on every PC, can be used to explore the optimization space at compile-time. Second, machine-learning techniques, e.g. Milepost project (`http://www.milepost.eu/`), can be used to automatically improve code generation compilers strategies. Speculation can be used to deal with necessary but missing information at compile-time. Finally, dynamic techniques can select or generate at run-time the most efficient code adapted to the execution context and available hardware resources.

Future compilers will benefit from past research, but they will also need to combine static and dynamic techniques. Moreover, domain specific approaches might be needed to ensure success.

Biography of François Bodin

François Bodin cofounded CAPS (www.caps-entreprise.com) in 2002 while he was a Professor at University of Rennes I and since January 2008 he joined the company as CTO. His contribution includes new approaches for exploiting high performance processors in scientific computing and in embedded applications. Prior to joining CAPS, François Bodin held various research positions at University of Rennes I and at the INRIA research lab. He has published over 60 papers in international journals and conferences and he has supervised over 15 PhD thesis. Professor François Bodin holds a Master's in CS and a PhD in CS, both from University of Rennes I.

Steal-on-Abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering

Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson

The University of Manchester
{ansari,mikel,kotselidis,jarvis,chris,watson}@cs.manchester.ac.uk

Abstract. In transactional memory, aborted transactions reduce performance, and waste computing resources. Ideally, concurrent execution of transactions should be optimally ordered to minimise aborts, but such an ordering is often either complex, or unfeasible, to obtain.

This paper introduces a new technique called *steal-on-abort*, which aims to improve transaction ordering at runtime. Suppose transactions A and B conflict, and B is aborted. In general it is difficult to predict this first conflict, but once observed, it is logical not to execute the two transactions concurrently again. In steal-on-abort, the aborted transaction B is stolen by its opponent transaction A, and queued behind A to prevent concurrent execution of A and B. Without steal-on-abort, transaction B would typically have been restarted immediately, and possibly had a *repeat conflict* with transaction A.

Steal-on-abort requires no application-specific information, modification, or offline pre-processing. In this paper, it is evaluated using a sorted linked list, red-black tree, STAMP-vacation, and Lee-TM. The evaluation reveals steal-on-abort is highly effective at eliminating repeat conflicts, which reduces the amount of computing resources wasted, and significantly improves performance.

1 Introduction

In the future, software will need to be parallelised to take advantage of the increasing number of cores in multi-core processors. Concurrent programming, using *explicit locking* to ensure safe access to shared data, has been the domain of experts, and is well-known for being challenging to build robust and correct software. Typical problems include data races, deadlock, livelock, priority inversion, and convoying. The move to multi-cores requires adoption of concurrent programming by the majority of programmers, not just experts, and thus simplifying it has become an important challenge.

Transactional Memory (TM) is a new concurrent programming model that seeks to reduce programming effort, while maintaining or improving execution performance, compared to explicit locking. TM research has surged due to the need to simplify concurrent programming. In TM, programmers mark those

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 4–18, 2009.

blocks of code that access shared data as *transactions*, and safe access to shared data by concurrently executing transactions is ensured *implicitly* (i.e. invisible to the programmer) by the TM implementation.

The TM implementation compares each transaction's data accesses against all other transactions' data accesses for conflicts. Conflicts occur when a transaction has a) read a data element and another transaction attempts to write to it, or b) written to a data element and another transaction attempts to read or write to it. If conflicting data accesses are detected between any two transactions, one of them is *aborted*, and usually restarted immediately. Selecting the transaction to abort, or *conflict resolution*, is based upon a policy, sometimes referred to as a *contention management policy*. If a transaction completes execution without aborting, then it *commits*, which makes its changes to shared data visible to the whole program.

Achieving scalability on multi-core architectures requires, amongst other things, the number of aborted transactions to be kept to a minimum. Aborted transactions reduce performance, reduce scalability, and waste computing resources. Furthermore, in certain (update-in-place) TM implementations aborted transactions require extra computing resources to roll back the program to a consistent state.

The order in which transactions concurrently execute can affect the number of aborts that occur, and thus affect performance. Although it may be possible to determine an optimal order (or schedule) that minimises the number of aborts given complete information a priori, in practice this is difficult to achieve. Often complete information is impractical to obtain, simply not available for some programs, e.g. due to dynamic transaction creation, or even if it is available, the search space for computing the optimal order may be unfeasibly large.

This paper presents a new technique called *steal-on-abort*, which aims to improve transaction ordering at runtime. When a transaction is aborted, it is typically restarted immediately. However, due to close temporal locality, the immediately restarted transaction may repeat its conflict with the original transaction, leading to another aborted transaction. Steal-on-abort targets such a scenario: the transaction that is aborted is not restarted immediately, but instead 'stolen' by the opponent transaction, and queued behind it. This prevents the two transactions from conflicting again.

Crucially, steal-on-abort requires no application-specific information or configuration, and requires no offline pre-processing. Steal-on-abort is implemented in DSTM2 [1], a Software TM (STM) implementation, and evaluated using a sorted linked list [2], red-black tree [2], STAMP-vacation [3], and Lee-TM [4,5]. The evaluation shows steal-on-abort to be highly effective at reducing repeat conflicts, which lead to performance improvements ranging from 1.2 fold to 290.4 fold.

The paper is organised as follows: Section 2 introduces steal-on-abort, its implementation in DSTM2, the different steal-on-abort strategies developed, and related work. Section 3 evaluates steal-on-abort, presenting results for transaction throughput, repeat conflicts, and briefly examining steal-on-abort overhead. Finally, Section 4 presents the conclusions.

2 Steal-on-Abort

In most TM implementations, aborted transactions are immediately restarted. However, we observed that the restarted transaction may conflict with the same opponent transaction again, leading to another abort, which we refer to as a *repeat conflict*. In general it is difficult to predict the first conflict between any two transactions, but once a conflict between two transactions is observed, it is logical not to execute them concurrently again (or, at least, not to execute them concurrently unless the repeat conflict can be avoided). Using steal-on-abort the aborter steals the abortee, and only releases its stolen transactions after committing. This prevents them from being executed concurrently, which reduces wasted work. However, steal-on-abort also aims to improve performance. When a transaction is abort-stolen, the thread that was executing it acquires a new transaction and begins executing it.

An advantage of steal-on-abort is that it complements existing contention management policies. Since steal-on-abort is only engaged upon abort, existing contention management policies can continued to be used to decide which transaction to abort upon conflict.

The remainder of this section explains the implementation of steal-on-abort in DSTM2, and then explores the steal-on-abort design space by suggesting several execution strategies. The implementation needs to support three key components of steal-on-abort. First, each thread needs to be able to store the transactions stolen by its currently executing transaction. Second, each thread needs to be able to acquire a new transaction if its current transaction is stolen. Finally, a safe mechanism for stealing active transactions is required.

2.1 Multiple Work Queue Thread Pool with Randomized Work Stealing

DSTM2, like other STMs [6,7,8], creates a number of threads that concurrently execute transactions. This is extended into a thread pool model, and application threads submit transactional jobs to a transactional thread pool. As shown in Figure 1, a work queue is added to each worker thread in the transactional thread pool (`java.util.concurrent.LinkedBlockingDeque`, a thread-safe deque) to store transactional jobs. A transactional job is simply an object that holds the information needed to execute a transaction (e.g., pointer to a function, and parameters). Multiple work queues are used as a single work queue would lead to high serialisation overhead, and submitted jobs are round robin distributed to work queues. Worker threads acquire transactions from the head of their own queue when their current transaction commits, or is abort-stolen.

In order to keep worker threads busy, randomised work stealing [9] is implemented as well. The terms *work-steal* and *abort-steal* are used to differentiate between transactions being stolen due to work stealing, and due to steal-on-abort, respectively. As shown in Figure 2, if a worker thread's own work queue is empty, it randomly selects another worker thread, and attempts to work-steal a single transactional job from the tail of that thread's work queue. If all

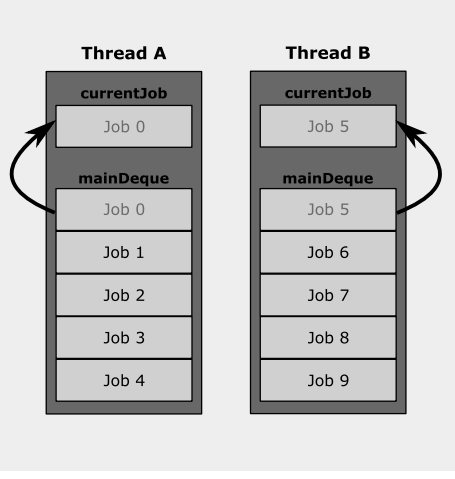


Fig. 1. Worker threads have per-thread deques that store transactional jobs. Worker threads take jobs from the head of their own deque.

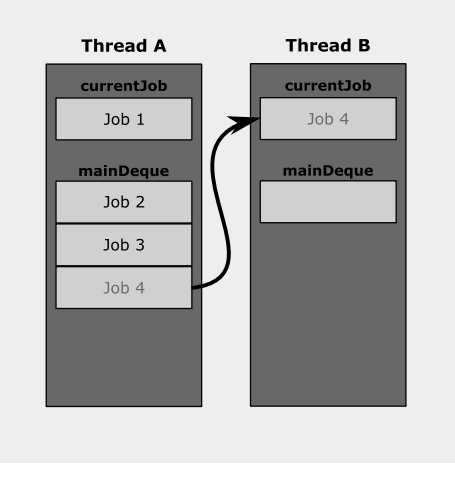


Fig. 2. If a worker thread's deque is empty, it work-steals a job from the tail of another randomly selected worker thread's deque

work queues are empty, the thread will attempt to work-steal from other worker threads' steal queues (described next).

2.2 Steal-on-Abort Operation

A private steal queue (also a `java.util.concurrent.LinkedBlockingDeque`) is added to each worker thread to hold transactional jobs that are abort-stolen,

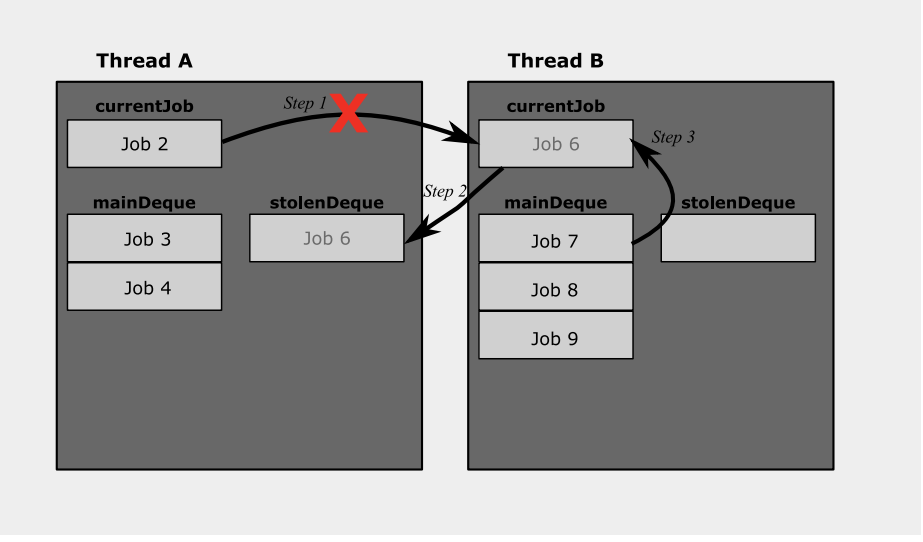


Fig. 3. Steal-on-abort in action. Worker thread A is executing a transaction based on Job 2, and worker thread B is executing a transaction based on Job 6. In step 1, thread A's transaction conflicts with, and aborts, Thread B's transaction. In step 2, thread A abort-steals thread B's job, and places it in its own steal queue. In step 3, after thread A finishes stealing, thread B gets a new job, and starts executing it.

as shown in Figure 3, which illustrates steal-on-abort in action. Each worker thread has an additional thread-safe flag, called `stolen`.

A stealing worker thread attempts to abort its victim worker thread's transaction. If this attempt is successful, the stealing thread takes the job stored in the victim thread's `currentJob` variable, and stores it in its own steal queue. After the job is taken, the victim thread's `stolen` flag is set. If a victim thread detects its transaction has been aborted, it waits for its `stolen` flag to be set. Once the flag is set, the victim thread obtains a new job, stores it in `currentJob`, and then clears the `stolen` flag. The victim thread must wait on the `stolen` flag, otherwise access to the variable `currentJob` could be unsafe.

2.3 Programming Model Considerations

There are two important programming model changes to consider when using a transactional thread pool. First, in our implementation, application threads submit transactional jobs to the thread pool to be executed asynchronously, rather than executing transactions directly. This requires a trivial change to the application code.

Secondly, application threads that previously executed a transactional code block, and then executed code that depended on the transactional code block (e.g. code that uses a return value obtained from executing the transactional code block), are not easily supported using asynchronous job execution. This

dependency can be accommodated by using synchronous job execution; for example, the application thread could wait on a condition variable, and be notified when the submitted transactional job has committed. Additionally, the transactional job object could be used to store any return values from the committed transaction that may be required by the application thread's dependent code. This requires a simple modification to the implementation described already. The use of asynchronous job execution, where possible, is preferred as it permits greater parallelism: application and worker threads execute simultaneously.

2.4 Steal-on-Abort Strategies

Four different steal-on-abort strategies constitute the design space investigated in this paper. Each strategy differs in the way stolen transactions are released. The properties of each one are explained below.

Steal-Tail inserts abort-stolen jobs at the tail of the `mainDeque` once the current transaction completes. This means the abort-stolen jobs will be executed last since jobs are normally taken from the head of the deque, unless other threads work-steal the jobs and execute them earlier. As mentioned earlier, jobs are created and distributed in a round-robin manner to threads' `mainDeque`s. Therefore, jobs created close in time will likely be executed close in time. Steal-Tail may benefit performance in a benchmark, for example, where a job's creation time has strong affinity with its data accesses, i.e. jobs created close in time have similar data accesses, which means they are likely to conflict if executed concurrently. Executing an abort-stolen job immediately after the current job may lead it to conflict with other concurrently executing transactions since they are likely to be those created close in time as well. Placing abort-stolen jobs at the tail of the deque may reduce conflicts by increasing the temporal execution distance between jobs created close in time.

Steal-Head inserts abort-stolen jobs at the head of the `mainDeque` once the current transaction completes. This means the abort-stolen jobs will be executed first. For benchmarks that do not show the affinity described above, placing jobs at the head of the deque may take advantage of cache locality and improve performance. For example, transaction A aborts and abort-steals transaction B. Upon completion of transaction A, transaction B is started. At least one data element is common to both transactions; the data element that caused a conflict between them, and is likely to be in the local cache of the processor (or core). The larger the data access overlap, the more likely performance is to improve.

Steal-Keep does not move abort-stolen jobs from a thread's `stealDeque` to its `mainDeque` once the current transaction completes. The thread continues to execute jobs from its `mainDeque` until it is empty, and then executes jobs from its `stealDeque` (when both are empty, work stealing is invoked as usual). The motivation of Steal-Keep is to increase the average time to an abort-stolen job's re-execution, as it will be executed last by the current thread, and only work-stolen by other threads if all other threads' `mainDeque`s are empty. Steal-Keep

may reduce steal-on-abort overhead as it does not require jobs to be moved from the `stealDeque` to the `mainDeque` after every transaction finishes, however, it may increase the overhead of work stealing when the `mainDeque` of all threads is empty.

Steal-Block causes an abort-stolen job's second-order abort-stolen jobs to be taken as well (thus a block of transactions is stolen). The hypothesis is that in some benchmarks there is a strong data access affinity between aborted transactions that extends further down the directed graph of aborted transactions. In such benchmarks, Steal-Block aims to give greater performance improvements by reordering transactions faster. However, it also increases steal-on-abort overhead, as on every steal-on-abort operation the `stealDeque` must be traversed to take the second-order abort-stolen transactions.

2.5 Limitations

There are two important limitations to steal-on-abort. First, steal-on-abort is only useful when repeat conflicts occur, as queueing transactions eliminates the chance of repeat conflicts. If an application has significant numbers of conflicts, but they are mostly unique conflicts, then the benefit of steal-on-abort may be reduced.

Second, in order to detect repeat conflicts, the TM implementation must support visible accesses, either read, write, or both. Using invisible reads and writes only allow conflicts to be detected between an active transaction and a committed transaction. Repeat conflicts require the detection of conflicts between two active transactions, as then one may abort, restart, and again conflict with the same opponent, if the opponent is still active.

2.6 Related Work

Research in transaction reordering for improving TM performance has been limited. Bai *et al.* [10] introduced a key-based approach to co-locate transactions based on their calculated keys. Transactions that have similar keys are predicted to have a high likelihood of conflicting, and queued in the same queue to be executed serially. Their implementation also uses a thread pool model with multiple work queues, but they do not support work-stealing or abort-stealing.

Although their approach improves performance, its main limitation is the requirement of an application-specific formula to calculate the keys. This makes their technique of limited use without application-specific knowledge, and performance is dependent on the effectiveness of the formula. For some applications it may be difficult to create effective formulae, and in the extreme case ineffective formula may degrade performance. In contrast, steal-on-abort does not require any application-specific information.

Our recent work [11] attempts to reduce aborts by monitoring the percentage of transactions that commit over a period of time. If the percentage is found to deviate from a specified threshold then worker threads are added or removed

from a transactional thread pool to increase or decrease the number of transactions executed concurrently. Although this work does not target repeat conflicts, it effectively schedules transactions to improve resource usage and execution performance.

Recent work by Dolev *et al.* [12], called CAR-STM, has similarly attempted to schedule transactions into queues based on repeat conflicts. CAR-STM also supports the approach by Bai *et al.* to allow an application to submit a function used to co-located transactions predicted to have a high likelihood of conflicting. Unlike steal-on-abort, CAR-STM does not support the use of existing contention management policies, does not implement work stealing to improve load balance and parallel performance, and does not investigate strategies such as those in Section 2.4.

Harris *et al.* [13] describe the retry mechanism, which allows an aborted transaction to block, and wait for the condition that caused it to abort to change, rather than restart immediately. However, retry must by explicitly called by the programmer, whereas steal-on-abort operates transparently.

3 Evaluation

The evaluation aims to investigate the performance benefits of the steal-on-abort strategies by executing several benchmarks using high contention configurations. In this section, the term *Normal* refers to execution without steal-on-abort. Steal-Tail, Steal-Head, Steal-Keep, and Steal-Block are abbreviated to Steal-T, Steal-H, Steal-K, and Steal-Blk, respectively. All execution schemes (including Normal) utilise the thread pool, and work stealing.

3.1 Platform

The platform used to execute benchmarks is a 4 x dual-core (8-core) Opteron 880 2.4GHz system with 16GB RAM, running openSUSE 10.1, and using Sun Hotspot Java VM 1.6 64-bit with the flags `-Xms4096m -Xmx14000m`. Benchmarks are executed using DSTM2 set to using the shadow factory, and visible reads. DSTM2 always uses visible writes. Although read and write visibility affect the amount of conflicts that occur, visible reads and writes are generally considered to give higher performance than invisible reads and writes when a large number of conflicts occur, and the benchmarks in this evaluation have large numbers of conflicts (see next). Experiments are executed with 1, 2, 4, and 8 threads, each run is repeated 9 times, and mean results are reported with ± 1 standard deviation error bars.

The published best contention manager (CM), called Polka [14], is used in the evaluation. Upon conflict, Polka waits exponentially increasing amounts of time for a dynamic number of iterations (equal to the difference in the number of read accesses between the two transactions) for the opponent transaction to commit, before aborting it. Polka's default parameters are used for controlling the exponentially increasing wait times [14].

3.2 Benchmarks

The benchmarks used to evaluate steal-on-abort are a sorted linked list, red-black tree, STAMP-vacation, and Lee-TM. Hereafter, they are referred to as List, RBTree, Vacation, and Lee-TM, respectively. Evaluating steal-on-abort requires the benchmarks to generate large amounts of transactional conflicts. Below, the benchmarks are briefly described, along with the execution parameters used to produce high contention.

List and RBTree transactionally insert or remove random numbers into a sorted linked list or tree, respectively. List and RBTree are configured to perform 20,000 randomly selected insert and delete transactions with equal probability. Additionally, after executing its code block, each transaction waits for a short delay, which is randomly selected using a Gaussian distribution with a mean duration of 3.2ms, and a standard deviation of 1.0. The delays are used to simulate transactions that perform extra computation while accessing the data structures. This also increases the number of repeat conflicts.

Vacation is a benchmark from the STAMP suite (version 0.9.5) ported to DSTM2. It simulates a travel booking database with three tables to hold bookings for flights, hotels, and cars. Each transaction simulates a customer making several bookings, and thus several modifications to the database. High contention is achieved by configuring Vacation to build a database of 128 relations per table, and execute 256,000 transactions, each of which performs 50 modifications to the database.

Lee-TM is a transactional circuit routing application. Pairs of coordinates for each route are loaded from a file and sorted by ascending length. Each transaction attempts to lay a route from its start coordinate to its end coordinate in a three-dimensional array that represents a layered circuit board. Routing consists of two phases: *expansion* performs a breadth-first search from the start coordinate looking for the end coordinate, and *backtracking* writes the route by tracing back a path from the end coordinate to the start coordinate. For high contention, the Lee-TM-t configuration [5] is used (i.e., no early release) with the `mainboard.txt` input file, which has 1506 routes. This input file has relatively long transactions, and a only minority of them cause contention so repeat conflicts should be limited in comparison to the other benchmarks. Furthermore, later transactions are more likely to conflict with each other because of large amounts of data accesses, and Steal-Blk may offer better performance in such conditions.

3.3 Transaction Throughput

Figure 4 illustrates the throughput results. Cursory observation shows that steal-on-abort always improves throughput over Normal execution, sometimes by significant margins. Performance variance is generally minimal between the steal-on-abort strategies compared to the difference with Normal, but Steal-Blk is less effective in Vacation, and slightly more effective in Lee-TM, while Steal-H is less effective in Lee-TM. Furthermore, Steal-K and Steal-T are the most consistent performers, and thus for brevity the discussion will mainly focus on the performance benefits of Steal-T.

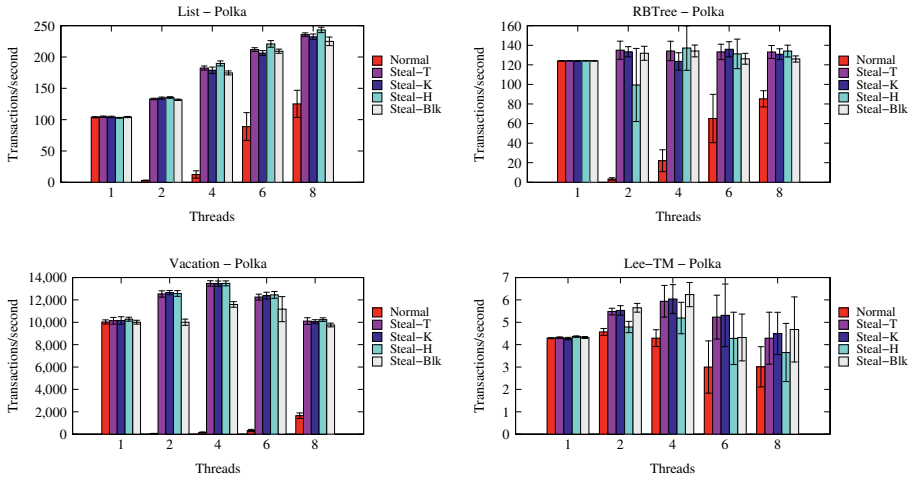


Fig. 4. Throughput results

In List, Steal-T improves average throughput over Normal by 46.7 fold with 2 threads, 14.6 fold with 4 threads, 2.4 fold with 6 threads, and 1.9 fold with 8 threads. Similarly, in RBTree the improvements are 40.0 fold, 6.1 fold, 2.0 fold, and 1.6 fold respectively. In Vacation the improvements are 290.4 fold, 92.9 fold, 37.9 fold, and 6.1 fold, respectively.

Examining Lee-TM, Steal-T improves average throughput over Normal by 1.2 fold with 2 threads, 1.4 fold with 4 threads, and 1.3 fold with 8 threads. However, Lee-TM results have high standard deviations, which increase with the number of threads. This is caused by Lee-TM performance being sensitive to the order in which transactions commit. As there are only 1506 routes, and most of the contention due to the long transactions executed near the end, even aborting a few long transactions in favour of other long transactions that have performed less computation can significantly impact performance. As predicted, Steal-Blk generally improves performance the most for Lee-TM.

3.4 Repeat Conflicts

Next, we examine the amount of time spent in repeat conflicts, and the effectiveness of the steal-on-abort strategies at reducing repeat conflicts. Figure 5 shows histograms of the distribution of wasted work [15] (i.e. the amount of time spent executing transactions that eventually aborted) for a given number of conflicts with the same transaction. As an example, consider a transaction A that is aborted seven times before it finally commits. Such a transaction has seven lots of wasted work. Four aborts occur through conflict with a transaction B, two with a transaction C, and one with a transaction D (seven in total). The four lots of wasted work caused by conflicting with, and being aborted by, transaction B are added to column '4', the two lots associated with C are added to column '2', and the one lot associated with D is added to column '1'. For

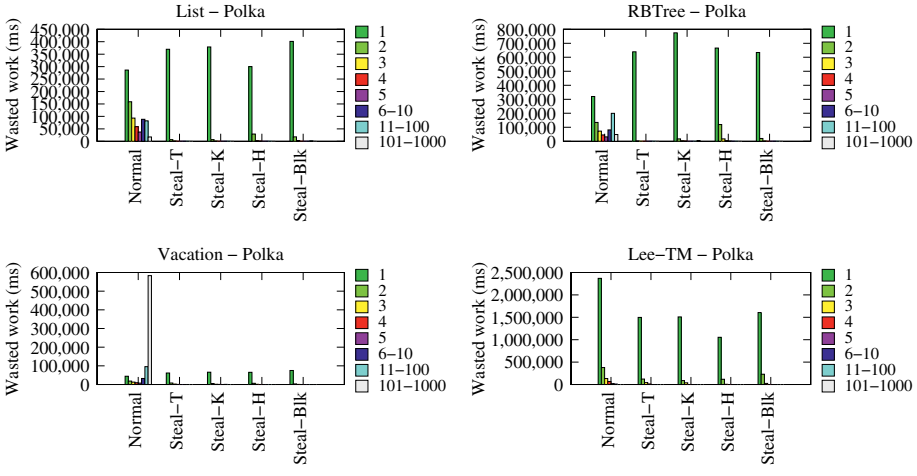


Fig. 5. Wasted work distribution by number of repeat conflicts

brevity, only results from execution with eight thread results are discussed, although better performance improvements were observed previously with fewer threads (Figure 4).

Since steal-on-abort should targets repeat conflicts it should reduce the amount of time in all but the first column. This is confirmed by the results in Figure 5: Steal-T reduces time in the remaining columns (repeat conflicts) by 99% in List, 95% in RBTree, 99% in Vacation, and 58% in Lee-TM. Furthermore, the results show that repeat conflicts represent a significant proportion of the total wasted work for the high contention configurations used: 65% in List, 54% in RBTree, 96% in Vacation, and 17% in Lee-TM. The net reduction in wasted work using Steal-T with 8 threads is 53% in List, 18% in RBTree, 93% in Vacation, and 13% in Lee-TM.

However, steal-on-abort increases single conflict (non-repeat) wasted work for List, RBTree, and Vacation. This is because repeat conflicts are being reduced to single conflicts so their wasted work is allocated to the single conflict column. However, the increase in single conflict wasted work is far less than the decrease in repeat conflict wasted work. As a result, Lee-TM, which has far fewer repeat conflicts than the other benchmarks, actually sees a fall in single conflict wasted work. Thus, a side effect of steal-on-abort is to reduce the number of single (i.e., unique) conflicts that occur.

3.5 Committed Transaction Durations

Polka causes transactions to wait for their opponents, which increases the average time it takes to execute a transaction that eventually commits. Since steal-on-abort reduced the amount of time spent in repeat conflicts, it should also have reduced the total number of conflicts, which in turn should have reduced the average committed transaction's duration.

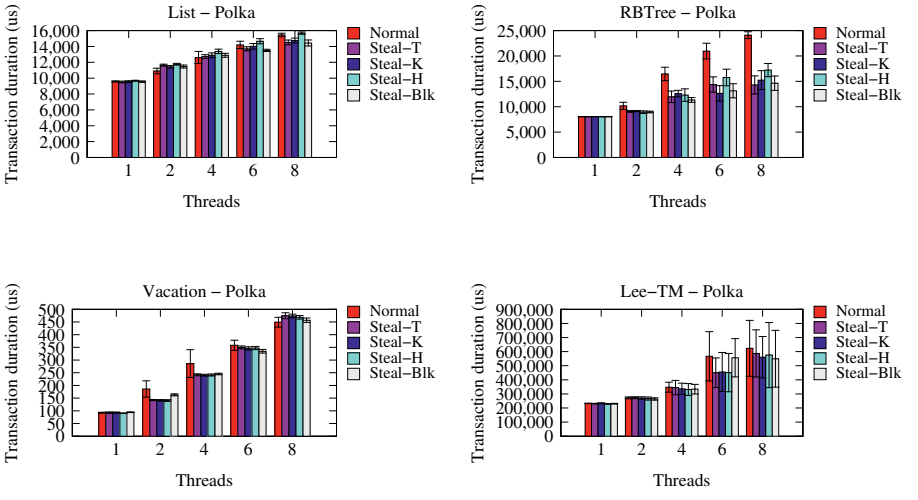


Fig. 6. Average Committed Transaction Duration (microseconds)

Figure 6 shows the results for the average committed transaction's duration, which includes the overhead of steal-on-abort operations, and confirms the hypothesis. Three of the four benchmarks reduce the average duration with 8 threads, except for List using Steal-H, which marginally increases the average duration. Only for Vacation do all the steal-on-abort strategies increase the average duration, although this is still largely within the standard deviations.

3.6 Steal-on-Abort Overhead

We have not precisely measured the overhead of steal-on-abort as it consists of small code blocks, some of which execute within transactions, and some outside of transactions. However, as shown in Figure 6, Vacation's transactions have much shorter average durations than the other benchmarks, and consequently Vacation's increase in average duration in Figure 6 may be due to abort-stealing overhead, which would indicate that the overhead is in the tens of microseconds per transaction.

However, this overhead does not include the cost of moving transactions between deques, as that happens after a transaction completes. To measure that cost the in-transaction metric (InTx), which is the proportion of execution time spent in executing transactions, is presented in Figure 7. For the benchmarks used in this evaluation there are two sources of out-of-transaction execution: work stealing, and moving jobs from a thread's `stolenDeque` to its `mainDeque` after every transaction completes. Since Normal execution utilises work stealing, the difference between Normal and steal-on-abort execution should approximately represent the cost of moving jobs between the deques.

Figure 7 identifies that there is negligible overhead in moving jobs between deques and work stealing in List, RBTree, and Lee-TM. However, in Vacation

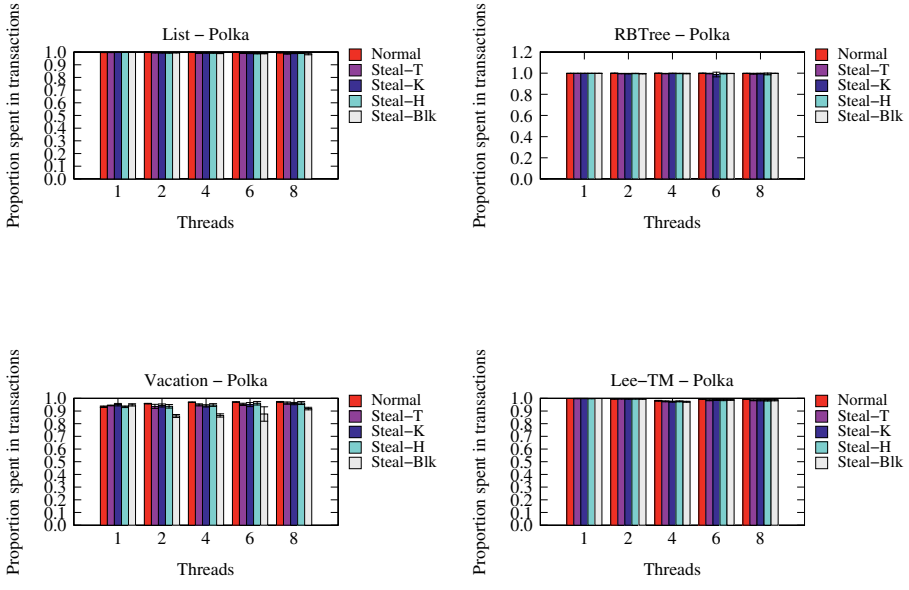


Fig. 7. Proportion of total time spent executing transactions

the overhead becomes visible, with most strategies observing an overhead of 3%. The average execution time of Vacation at 8 threads with Steal-T is 24.0 seconds, and given that 256,000 transactions are executed, the average overhead of moving jobs is 2.8 microseconds per transaction. However, this cost is related to the number of jobs moved between deques, and with Steal-T this averages to 2.2 jobs per completed transaction. Section 2.4 mention that Steal-Blk may have higher overhead, and Vacation's results identify Steal-Blk observing an overhead of 5-10%.

4 Conclusions and Future Work

In well-engineered, scalable, concurrently programmed applications it is expected that high contention conditions will occur only rarely. Nevertheless, when high contention does occur it is important that performance degrades as little as possible. It is also probable that some applications will not be as well-engineered as expected, and thus may suffer from high contention more frequently.

This paper presented steal-on-abort, a new runtime approach that dynamically reorders transactions with the aim of improving performance by reducing the number of repeat conflicts. Steal-on-abort is a low overhead technique that requires no application specific information or offline pre-processing.

Steal-on-abort was evaluated using the well-known Polka contention manager with two widely used benchmarks in TM: sorted linked list and red-black tree, and two non-trivial benchmarks: STAMP-vacation and Lee-TM. The benchmarks were configured to generate high contention, which led to significant

amounts of repeat conflicts. Steal-on-abort was effective at reducing repeat conflicts: Steal-Tail reducing by almost 60% even when repeat conflicts only accounted for 17% of the total wasted work, and reducing by over 95% when repeat conflicts accounted for 55% or more of the wasted work. This led to performance improvements ranging from 1.2 fold to 290.4 fold.

We are encouraged by the results from the steal-on-abort evaluation, and we plan to continue our investigation of the design space. In particular, we wish to investigate the design of steal-on-abort when invisible reads and writes are used, and the implementation of steal-on-abort for HTMs.

References

1. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. In: OOPSLA 2006: Proceedings of the 21st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 253–262. ACM Press, New York (2006)
2. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC 2003: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing, pp. 92–101. ACM Press, New York (2003)
3. Minh, C.C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K.: An effective hybrid transactional memory system with strong isolation guarantees. In: ISCA 2007: Proceedings of the 34th Annual International Symposium on Computer Architecture, pp. 69–80. ACM Press, New York (2007)
4. Watson, I., Kirkham, C., Luján, M.: A study of a transactional parallel routing algorithm. In: PACT 2007: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques, pp. 388–400. IEEE Computer Society Press, Los Alamitos (2007)
5. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Lee-TM: A non-trivial benchmark for transactional memory. In: Bourgeois, A.G., Zheng, S.Q. (eds.) ICA3PP 2008. LNCS, vol. 5022, pp. 196–207. Springer, Heidelberg (2008)
6. Marathe, V., Spear, M., Herio, C., Acharya, A., Eisenstat, D., Scherer III, W., Scott, M.L.: Lowering the overhead of software transactional memory. In: TRANSACT 2006: First ACM SIGPLAN Workshop on Transactional Computing (June 2006)
7. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
8. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 237–246. ACM Press, New York (2008)
9. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Journal of Parallel and Distributed Computing 37(1), 55–69 (1996)

10. Bai, T., Shen, X., Zhang, C., Scherer, W.N., Ding, C., Scott, M.L.: A key-based adaptive transactional memory executor. In: IPDPS 2007: Proceedings of the 21st International Parallel and Distributed Processing Symposium. IEEE Computer Society Press, Los Alamitos (2007)
11. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Advanced concurrency control for transactional memory using transaction commit rate. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 719–728. Springer, Heidelberg (2008)
12. Dolev, S., Hendler, D., Suissa, A.: Car-stm: Scheduling-based collision avoidance and resolution for software transactional memory. In: PODC 2007: Proceedings of the 26th annual ACM symposium on Principles of distributed computing (August 2008)
13. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP 2005: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 48–60. ACM, New York (2005)
14. Scherer III, W., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC 2005: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing, pp. 240–248. ACM Press, New York (2005)
15. Perfumo, C., Sonmez, N., Cristal, A., Unsal, O., Valero, M., Harris, T.: Dissecting transactional executions in Haskell. In: TRANSACT 2007: Second ACM SIGPLAN Workshop on Transactional Computing (August 2007)

Predictive Runtime Code Scheduling for Heterogeneous Architectures*

Víctor J. Jiménez[1], Lluís Vilanova[2], Isaac Gelado[2], Marisa Gil[2], Grigori Fursin[3], and Nacho Navarro[2]

[1] Barcelona Supercomputing Center (BSC)
victor.javier@bsc.es
[2] Departament d'Arquitectura de Computadors (UPC)
{igelado,vilanova,marisa,nacho}@ac.upc.edu
[3] ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud University
grigori.fursin@inria.fr

Abstract. Heterogeneous architectures are currently widespread. With the advent of easy-to-program general purpose GPUs, virtually every recent desktop computer is a heterogeneous system. Combining the CPU and the GPU brings great amounts of processing power. However, such architectures are often used in a restricted way for domain-specific applications like scientific applications and games, and they tend to be used by a single application at a time. We envision future heterogeneous computing systems where all their heterogeneous resources are continuously utilized by different applications with versioned critical parts to be able to better adapt their behavior and improve execution time, power consumption, response time and other constraints at runtime. Under such a model, adaptive scheduling becomes a critical component.

In this paper, we propose a novel predictive user-level scheduler based on past performance history for heterogeneous systems. We developed several scheduling policies and present the study of their impact on system performance. We demonstrate that such scheduler allows multiple applications to fully utilize all available processing resources in CPU/GPU-like systems and consistently achieve speedups ranging from 30% to 40% compared to just using the GPU in a single application mode.

1 Introduction

The objectives of this work are twofold. On the one hand, fully exploiting the computing power available in current CPU/GPU-like heterogeneous systems and thus, increasing overall system performance is pursued. On the other hand, exploring and understanding the effect of different scheduling algorithms for heterogeneous architectures is intended.

Currently almost every desktop system is an heterogeneous system. They both have a CPU and a GPU, two processing elements (PEs) with different characteristics but undeniable amounts of processing power. Some time ago programming

* All the authors are members of the HiPEAC European Network of Excellence.

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 19–33, 2009.

a GPU for general purpose computations was a major programming challenge. However, with the advent of GPUs designed with general purpose computation in mind it has become simpler. Games still represent the big market for graphical card manufacturers, but thanks to execution models like CUDA [2] now it is possible to use such GPUs as data parallel computing devices.

Applications with great amounts of data parallelism perform considerably better on a GPU than on a CPU. [13] Thus, the GPU is seen as a device destined to run very specific workloads. The current trend to program these systems is as following: (1) Profile the application to be ported to a GPU and detect the most expensive parts in terms of execution time and the most amenable ones to fit the GPU-way of computing (i.e., data parallelism). (2) Port those code fragments to CUDA kernels (or any other framework for general purpose programming on GPU). Getting peak performance from GPUs is not extremely easy, so this can be initially done in a fast way, despite not being very optimal. (3) Iteratively optimize the kernels until the desired performance is achieved.

However, in the authors' opinion, those architectures are currently being used in a somewhat restricted way by domain-specific applications such as scientific applications and games, and tend to be used by a single application at a time. The authors envision future heterogeneous computing systems where all their PEs are continuously utilized by different applications with versioned critical parts to be able to better adapt their behavior and improve execution time, power consumption, response time and other constraints at runtime. Under such a model, adaptive scheduling becomes a critical component.

The proposal presented here consists of a novel predictive user-level scheduler based on past performance history for heterogeneous systems. Several scheduling policies have been developed and the study of their impact on system performance is presented. Such scheduler allows multiple applications to fully utilize all the PEs in CPU/GPU-like systems and consistently achieve speedups ranging from 30% to 40% compared to just using the GPU in a single application mode.

This paper is structured as follows. In section 2 the scheduling framework is introduced. Section 3 describes the experimental methodology followed for this work. In section 4 the performance results for the scheduler are presented. Section 6 concludes the paper and lists potential future work.

2 Code Scheduling

The heterogeneous scheduling system lies at the core of this work. This section explains in detail its design and implementation. Additionally, a usage example and some guidelines for the future are provided.

2.1 Scheduler Design

For this study, a function-level granularity for code versioning and scheduling is used. Currently, a programmer has to indicate that a function can be executed on both CPU and GPU providing explicit data transfer if needed. Considering this study is performed on top of real hardware with a multi-ISA architecture,

it does not seem feasible to use a granularity finer than function-level. Indeed, that level seems a good choice for the programmer to provide both versions of the code (CPU and GPU). It it important to mention that the generation of function versions for every PE is orthogonal to this work and it is expected for future compilers to be able to generate multiple versions which can be adaptively used at runtime. [6,7] Additionally, there are already studies which allow to automatically generate a function version for one PE given the version for another PE. [16]

The code scheduler has been implemented as a dynamic library for the GNU/Linux OS. Being a process-level scheduler, parts of the library must be shared among all the processes which use it (see Figure 1). Specifically the data for the PE management and the task queues for each PE are shared (a task, composed by a function and its arguments, is used as the basic unit of scheduling from now on in this text). Other implementation options such as creating a kernel-level scheduler have been considered. However, it poses many difficulties, involving a longer development cost and the necessity to deal with NVIDIA's proprietary driver. Being simpler and, at the same time, enough to perform this study, the implementation uses the dynamic library approach. The interface to the scheduler is a set of C++ classes.

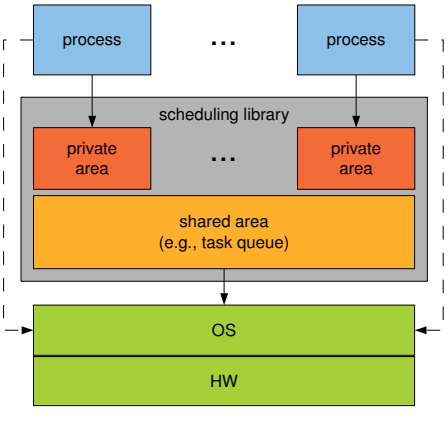


Fig. 1. Call scheduler implementation overview

2.2 Usage

A matrix multiplication is used as an example of the scheduler usage. A typical implementation would provide a function, matrix_mul, which implements the matrix multiply operation. This function would be called with two input matrices and one output matrix. Additionally, the matrix size would be also provided:

```
matrix_mul(A,B,C,N);
```

If this function must run on several PEs, multiple implementations are necessary. In the case of a CPU and a GPU they could be named matrix_mul_cpu

and matrix_mul_gpu. Considering this is already done, the interaction with the scheduler is quite simple. The user would get an instance of the main scheduler class (CallScheduler). Then the user constructs a call (Func, Args) and executes the schedule method. This creates a Task which is added to the queue for the PE selected by the scheduling algorithm. Different algorithms can be plugged-in in the scheduling system, thus making the system very flexible for trying new scheduling algorithms. Following is the extra code necessary to be able to let the scheduler select at runtime which PE will be used to perform the operation:

```
CallScheduler* cs = CallScheduler::getInstance();
MatrixMulFunc* f = new MatrixMulFunc();
MatrixMulArgs* a = new MatrixMulArgs(A,B,C,N);
cs->schedule(f,a);
```

where class MatrixMulArgs is a very simple class which just stores the values for the arguments to the function and MatrixMulFunc is a wrapper which allows to select the right function version to execute in a given PE. This is easily doable because CUDA stores both versions in the same executable file. In our implementation just a call to either matrix_mul_cpu or matrix_mul_gpu is necessary. Although it may seem a considerable amount of code, it is possible to use some "syntactic sugar" which would allow, for instance, a source-to-source compiler to generate all that code from a line similar to:

```
#pragma cs matrix_mul(A,B,C,N) matrix_mul_cpu matrix_mul_gpu
```

2.3 Scheduling Algorithms

In a heterogeneous scheduling process the following two steps can be distinguished: *PE selection* and *task selection*. The former is the process to decide on which PE a new task should be executed. It does not mean the execution is going to start at that time. The latter it is the mechanism to choose which task must be executed next in a given PE. It typically takes place just after another task finishes and its PE becomes free.

Several options have been tested for the first step. All the algorithms basically follow a variant of the *first-free* (FF) design, meaning that tasks are first tried to be scheduled in a PE which is not being used. As the results will show, this approach does not work consistently good all the time and thus, new algorithms based on *performance-prediction* have been developed.

For the second step, all the algorithms implemented in this work follow a *first-come, first-served* (FCFS) design. It could be also possible to implement some more advanced techniques such as work stealing in order to increase the load balance of the different PEs. However, the main goal of the study was to find algorithms which led to a good scheduling depending on the code to be executed and the characteristics of the PEs present in the system. Thus, the study for load balancing techniques is left for future work.

Several variants of different families of algorithms have been developed. The following description gives the general scheme for these families. The specific parts for every variant are abstracted as calls to functions (g and h).

Algorithm 1 shows the general scheme for a FF design. It traverses the PE list in search for a not busy one. As soon as one is found it is selected as the target PE. If none is idle the algorithm must decide which PE to use. Several variants have been tried and thus the algorithm contains a call to a function g which will be responsible to select somehow a PE in case all of them are busy.

Algorithm 1. First-Free algorithm family

```
for all pe ∈ PElist do
  if pe is not busy then
    return pe
return g(PElist)
```

As the CPU and the GPU present different characteristics, the same function may perform differently in both PEs. It could be the case that one of them performs better for some kind of tasks. Therefore a modification to the previous algorithm is introduced, allowing to queue more elements into one PE, thus introducing a bias in the scheduling system. This can also be seen as a simple load balancing mechanism. Algorithm 2 is still *first-free*-based, but in case all the PEs are busy it will assign tasks to PEs following a distribution given by a parameter $k = (k_1, \ldots, k_n)$. Given two PEs, a and b, the ratio k_a/k_b determines the amount of work which will be given to them. For instance, with $k = (1, 4)$ the number of tasks given to the second PE will be four times bigger.

Algorithm 2. First-Free Round Robin (k)

```
for all pe ∈ PElist do
  if pe is not busy then
    return pe
if k[pe] = 0 ∀pe ∈ PElist then
  set k with initial values
for all pe ∈ PElist do
  if k[pe] ≥ 0 then
    k[pe] = k[pe] − 1
    return pe
```

The idea behind this algorithm is that if a set of applications is biased towards one of the PEs, consistently obtaining better performance on it, the scheduler may address the load imbalance by biasing the assignment towards the other PE. However it may also happen that performance for an application drastically differs depending on the PE where it is run. In those cases the previous algorithm may not perform well. This observation motivated the introduction of a *performance history*-based scheduler (algorithm 3). Basically a performance history is kept for every pair of PE and task. During the initial phase, the performance history is built by forcing the first n calls to the same function to execute on the different n PEs. In the next phase every time a call to that function is made,

the scheduler looks for any big unbalance between the performance on the different PEs. Thus, a list ($allowed_{PE}$) is built where only the PEs without such a big unbalance are kept. If that list came to be empty, g would determine which PE to select. h performs the corresponding action when there is more than one possibility to schedule the task.

Algorithm 3. Performance History Scheduling

```
if ∃pe ∈ PElist : history[pe, f] = null then
   return pe
allowedPE = {pe| ∄pe' : history[pe, f]/history[pe', f] > θ}
if ∃pe ∈ allowedPE : pe is not busy then
   return pe
if allowedPE = ∅ then
   return g(PElist)
else
   return h(allowedPE)
```

Relying on this performance prediction mechanism, a variant of algorithm 3 has been developed. It uses the performance history to predict the waiting time for every PE. This version aims at a better load balancing among the PEs.

3 Experimental Methodology

The runtime CPU/GPU scheduler has been evaluated on a real machine with a set of benchmarks. In the following subsections the benchmarks will be described in detail as well as the experimental setup.

3.1 Workload

A mix of synthetic and real benchmarks have been used in order to evaluate the performance speedup obtained with the use of the runtime code scheduler for the CPU/GPU system. The benchmarks used are: matmul, ftdock, cp and sad. Their performance characterization can be seen in table 1.

Table 1. Benchmark list characterization

Benchmark	CPU	GPU	Speedup	TX time	Comp Time	Ratio
cp	28.79s	0.39s	74X	0.13s	0.14s	1.08
sad	0.79s	0.87s	$\sim$ 0.9X	0.11s	0.04s	0.36
FTDock	38.77s	19.99s	$\sim$ 1.9X	$\sim$ 0.03s	0.34s	11.75
matmul	38.52s	12.65s	3X	0.01s	0.04s	3.89

matmul is a synthetic benchmark which performs multiple square-matrix multiplications using either the ATLAS library [21] for the CPU and the CUBLAS library [2] for the GPU. As can be seen in table 1, performance on GPU does not

extremely differ from performance on CPU. As the input size is increased, the GPU can better amortize the cost of bringing data in and out to main memory. The matrices used are considerable large (1024×1024).

FTDock [8] is a real application which computes the interaction between two molecules (docking). FFTW [5] is used in order to speedup this process. A hybrid version has been developed allowing to execute any of the rotations either on the CPU or the GPU. NVIDIA's CUFFT library [2] is used for the GPU. The changes introduced in the program are minimal since both libraries have almost the same interface. Although the GPU version runs twice as fast, the difference with the CPU is not big.

The Parboil Benchmark Suite [1] is a set of benchmarks designed to measure the performance of a GPU platform. They are available from the Impact Research Group at University of Illinois (UIUC). The benchmarks used here are cp and sad. cp computes the coulombic potential at each grid point over on plane in a 3D grid in which point charges have been randomly distributed. sad is used in MPEG video encoders in order to perform a sum of absolute differences between frames. While sad performs almost equal on both PEs, cp does so much more efficiently on the GPU. The application speedup is really large, thus, it is really crucial to schedule the application to the right PE (the GPU in this case). Due to some constraints, such as a big memory footprint for some of the others benchmarks included in the suite, only those two benchmarks have been used to evaluate the scheduling system. The GPU has a limited amount of memory, but a recent work proposes an architecture which would remove this constraint allowing to use virtual memory from the coprocessor (the GPU in this case) [9].

3.2 Experimental Setup

All the experiments have been run on real hardware. A machine with an Intel Core 2 E6600 processor running at 2.40GHz and 2GB of RAM has been used. The GPU is an NVIDIA 8600 GTS with 512MB of memory. The operating system is Red Hat Enterprise Linux 5.

The execution of the benchmarks is organized as combinations of N benchmarks running in parallel. In order to evaluate the scheduling system, different values of N have been tried. The amount of memory on the GPU limits how many processes can be concurrently run on it. Therefore, the values selected for N are $N = \{4, 6\}$. In order to keep experimenting time under reasonable values, fifteen randomly selected combinations have been chosen from all the possible permutations. In early tests performed with all the executions it was not possible to observe any significant change in the results, so the number selected seems a good compromise between results accuracy and experimenting running time.

As a way to improve results accuracy, experiments are repeated a small number of times and the results are averaged. The results obtained show that variance is not very high between executions ($< 6\%$ for the prediction-based algorithms).

4 Evaluation

Every experiment is composed of the execution of several combinations of benchmarks. Short names for the benchmark combinations are in the X-axis for some of the following figures (first letter for every benchmark name is used).

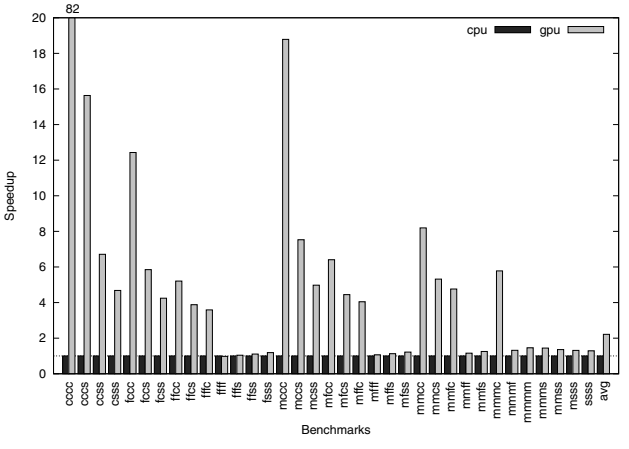


Fig. 2. CPU versus GPU performance

Figure 2 shows the performance obtained when running the combinations of benchmarks on just the CPU or the GPU. The GPU performs much better in around half of the cases and slightly better in the rest, with just one case being the other way around. Considering most of the benchmarks fit quite well the data-level parallelism paradigm, it is not surprising to see those results. Although it would be interesting to study the effect of a more balanced set of benchmarks, where some of the benchmarks performed better in the CPU than the GPU, this situation is still very interesting for the study. As some of the benchmarks perform much better on a specific PE, the scheduler must be very careful about its decisions in order not to damage performance. It will be shown that by using performance prediction algorithms the risk of scheduling a task on a very poor performing PE can be greatly reduced.

For the rest of the performance graphs no bar for the CPU is plotted since its performance is consistently worse than the one for the GPU. Therefore, the baseline for speedup graphs is the performance of the GPU alone executing the benchmarks serially (Figure 2). A trivial algorithm like a plain first-free one could also be used as the baseline as well. However, current systems do not allow the user to schedule tasks either on the CPU or the GPU, and thus it would not reflect the real benefit against current ways of CPU/GPU-systems utilization.

In general the benchmarks run using the heterogeneous scheduler achieve a considerable speedup compared to running them on a single PE. As can be seen in Figures 3 and 4, the average speedup obtained compared to the baseline is

between 30% and 40% (individual speedups for each combination are averaged using the harmonic mean). This is obviously a noticeable speedup which confirms to be worth to consider all the PEs in the system as resources where code can be scheduled for execution.

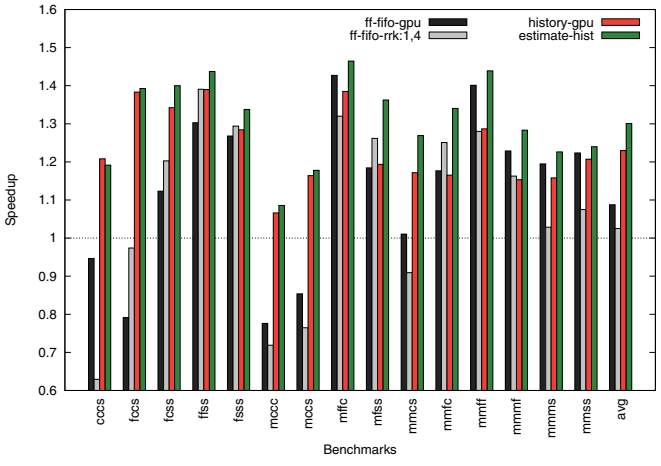


Fig. 3. Performance speedup for $N = 4$

4.1 First-Free Algorithms

Figures 3 and 4 show the relative speedup compared to the GPU for two first-free algorithm variants (ff-fifo-gpu and ff-fifo-rrk:1,4). They work as described in algorithms 1 and 2 in section 2. In case both PEs are busy, the first algorithm chooses the GPU, while the second schedules four tasks to the GPU for each one scheduled to the CPU in a round-robin way.

Despite being considerably simple, these algorithms perform well enough for some cases, reaching up to a 60% speedup over the baseline for a specific case (ffffffc in Figure 4). However, they are quite sensible to heavily-biased tasks. If a task performs much better on one PE than in the other ones, scheduling it on the wrong PE will considerably degrade overall system performance. This can be seen for instance in Figure 4 for the benchmark combination mmcccs. That combination contains three times the benchmark cp, which is strongly biased towards the GPU.

As first-free algorithms blindly select a PE without taking into account the characteristics of the PEs and the task to be scheduled, they eventually schedule cp to the CPU, resulting in a dramatic loss of performance. This behavior can be observed in Figure 6, where the distribution of PEs is shown for every benchmark and scheduling algorithm. The figure shows how few executions of cp are placed on the CPU. Even such a small percentage can greatly reduce performance, and if cp is not scheduled to the CPU more times is because by the time cp executions on the CPU finish, the rest of the benchmarks have already finished, leaving the GPU free and allowing cp to be scheduled on the GPU the rest of the time.

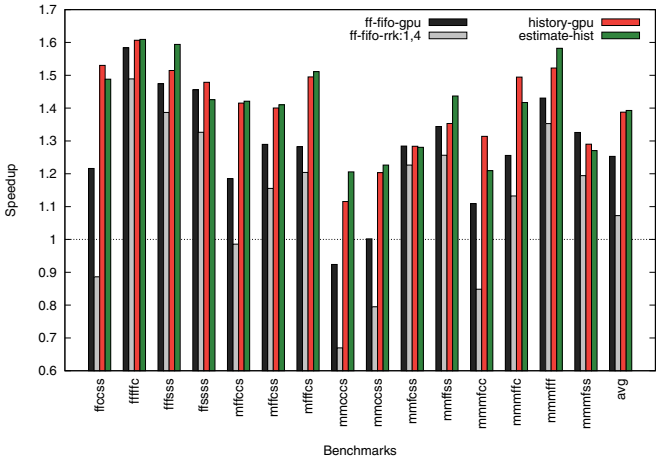


Fig. 4. Performance speedup for $N = 6$

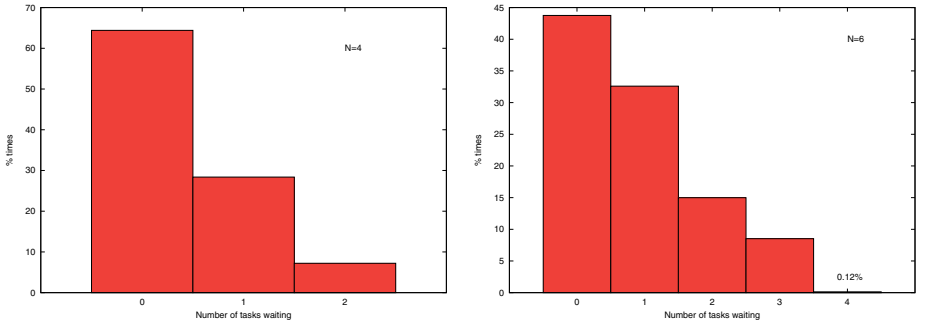


Fig. 5. Task queue usage histogram

Nonetheless, despite being ill-suited for workloads with highly-biased tasks, these algorithms perform quite acceptable in average, reaching around 10% speedup for $N = 4$ and 20% for $N = 6$. Other FF variants have been evaluated but they are not listed here since their performance was typically worse.

4.2 Predictive Algorithms

The algorithms studied here are two: history-gpu and estimate-hist. Both are based on algorithm 3 described in section 2. They keep the performance history for every pair (task, PE) in such a way that it is not allowed to schedule a task on a PE if the performance ratio to all the other PEs is worse than a threshold θ. Experimentally, the value $\theta = 5$ has been selected.

Both algorithms determine the set of PEs which can be used to schedule a task, $allowed_{PE}$. After that, history-gpu looks for the first idle PE in that set and

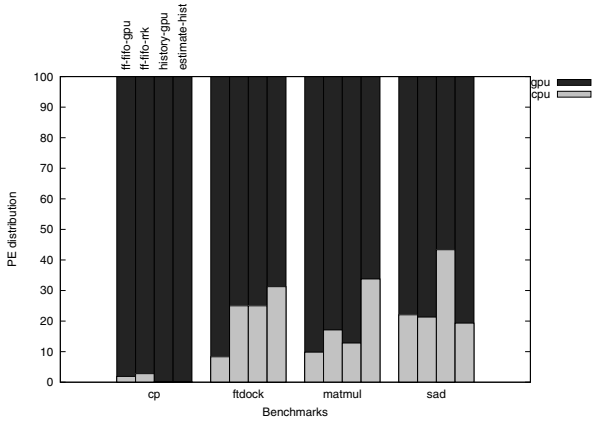


Fig. 6. PE distribution for different benchmarks and scheduling algorithms

if there is not such a PE, it selects the GPU as the target. A more fair version which schedules a task randomly to any of the PEs in $allowed_{PE}$ has also been evaluated, but the performance was not as good and results are not shown here.

The behavior of estimate-hist after $allowed_{PE}$ has been computed is quite different. It basically estimates the waiting time in each queue and schedules the task to the queue with the smaller waiting time (in case both queues are empty it chooses the GPU). For that purpose the scheduler uses the performance history for every pair (task, PE) to predict how long is going to take until all the tasks in a queue complete their execution.

Both algorithms achieve significant speedups compared to the baseline. For $N = 4$ estimate-hist has around a 30% speedup and history-gpu obtains a 20% speedup. Those speedups become bigger when $N = 6$, reaching almost a 40% in both cases. As can be seen in the Figures 3 and 4 both algorithms perform consistently well across all benchmark combinations compared to first-free algorithms. The main reason for that is the proper scheduling of cp to the GPU. However there seem to be other factors as well, as for some combinations of benchmarks where cp is not appearing (for instance, mmmfff in Figure 4) these algorithms, and especially estimate-hist, perform noticeably better than first-free based ones. The reason for that is that estimate-hist manages to better balance CPU and GPU task queues. This observation can be seen in Figure 6 where both predicting algorithms tend to schedule a higher percentage of the total number of tasks executed on the system to the CPU. Obviously, this is done without falling into scheduling a strongly-biased benchmark, as cp, to the CPU.

One interesting thing to note is the relatively poor results of estimate-hist for $N = 6$, not being able to improve history-gpu performance as much as in the $N = 4$ case. Due to the non-deterministic nature of scheduling there may not be just a single explanation for this effect. However, it has been observed how the prediction accuracy decreases by more than a 10% when the number of concurrent tasks increases from $N = 4$ to $N = 6$. estimate-hist, having two levels

of predictions, is more affected by this loss of accuracy than history-gpu. In order to improve the analysis this effect it would be interesting to conduct new tests on, for instance, quad-core machines where one or two cores can be freed from executing tasks, thus reducing possible interferences.

4.3 Effect of the Number of Tasks on the Scheduler

The number of simultaneous benchmarks run in an experiment is denoted by the value of N. As N increases, the number of tasks which compete for execution raises as well. This effect can be seen in Figure 5, which shows the number of waiting tasks that are in the queue every time a new task is being scheduled on a PE. Left graph is for the case where there are four benchmarks running at the same time, whereas right one depicts the case for six. The number of tasks waiting at the queues substantially increases from one case to the other because more processes are simultaneously using the scheduler.

If the number of tasks to be scheduled increases means it is possible to get closer to fully use all the PEs in the system. Thus, the number of times that the PEs are idle is reduced. Theoretically this must improve the throughput, as can be seen in Figures 3 and 4. When running four benchmarks a speedup of around 30% is achieved, whereas for the case of six the speedup is around 40%.

However, increasing the number of tasks increases the pressure on all the PEs. While on the GPU only tasks are being run, on the CPU parts of the benchmarks not corresponding to tasks and other processes such as Linux system processes are run at the same time as well. Increasing the number of tasks running on the CPU leaves less processing power for non-task parts and in some cases the overall performance may degrade.

In the future, and especially considering new processors with 4 and 8 cores, it may be worth reserving some cores for non-task computations, in order to decrease the interferences between task execution and other processes.

5 Related Work

Job and resource scheduling is a vastly explored topic. However, it seems just a few studies target scheduling for heterogeneous architectures. It is true that heterogeneity has been present in many scheduling studies, but it was mainly from the perspective of distributed and grid systems. The area of scheduling for heterogeneous architectures (i.e., within a single machine) is relatively new and has not been studied in detail yet. Some of the few papers on this topic [12,19] also agree on this lack of studies.

Scheduling for heterogeneous distributed systems is somehow similar to the problem being dealt here. PEs across different machines or within a single machine are heterogeneous and thus, they present different performance characteristics. However, in the case of distributed systems, the scheduling is burdened with many more complexities such as interprocessor communication (typically done across some kind of external network), distributed management and variant amount of

computing resources (some resources may disappear suddenly, whereas others can turn up) [18,3].

Most of the studies related to heterogeneous distributed systems propose new scheduling algorithms in order to improve the performance through job and resource allocation. They mathematically represent a program as a graph where every node is a task which can be mapped on a PE. Tasks which depend on each other are connected in the graph with the edge weight meaning the communication cost. The list of nodes is generated at compile time and the cost of running tasks on every PE is known in advance. Some relevant works on this area are [20,11,14,15]. However, communication cost in heterogeneous multicore systems is several orders of magnitude smaller than in distributed systems.

A few papers [12,19] do study the effect of scheduling for heterogeneous architectures. Similarly to the schedulers for distributed systems, programs are also represented as graphs with nodes corresponding to tasks in the program using information known a priori. In order to conduct performance measurements they mainly use random graphs as the input for the scheduler. This is one of the main differences compared to this work, where all the measurements are conducted on real hardware with a real software implementation and real applications.

In [4] a runtime scheduling system for the IBM Cell processor which eases application partitioning among different PEs is presented. However, in that work the heterogeneous architecture is viewed in the common way where the coprocessors are responsible for executing the computationally expensive parts while the main PE is just used to control the coprocessors.

As far as the authors know, this is the first work which deals with scheduling for a heterogeneous architecture using a real implementation and considering the system as a pool of PEs, being able to schedule tasks to any PE.

Initially a set of "classical" scheduling algorithms have been used. FF (first-free), FCFS (first come, first served), SJF (shortest job first) and RR (round-robin) are very well-known algorithms. Because of its simplicity and its relatively good performance, several variations of the FF algorithm have been tested in this study. It is difficult to track down the origin of those algorithms, so the reader is referred to Tanenbaum's work [17] for their description.

6 Conclusions and Future Work

This work shows how using a predictive scheduler for a CPU/GPU-like heterogeneous architectures can improve overall system performance. By adaptively scheduling versioned functions at run-time we can obtain speedups as high as 40% on average compared to perform the computation serially on the GPU.

Some specific applications achieve a large speedup when executed on a data-processing architectures such as a GPU. For these applications, with speedups over $100X$, it may not be worth to execute computationally expensive parts of them on the CPU. However, as this study demonstrates, there are other applications which can greatly increase performance by using a system like the one presented here.

Different kinds of scheduling algorithms have been tried. *first-free*-based ones perform noticeably well for some cases; however they fail to do so for biased computations where one PE performs much better than the others. Performance predicting algorithms, being able to avoid these cases and better balancing the system load, perform consistently better. We intend to study new algorithms in order to further improve overall system performance. Additionally, other benchmarks with different characteristics will be also tried. We expect that with a more realistic set of benchmarks (not only GPU-biased) the benefits of our system would be increased.

The results show how the tasks can receive interferences from other computation occurring in the system. Exploring how the number of cores present in the CPU affect that interference is an interesting future work. Additionally, a way to couple (or even merge) the scheduler presented here with the OS scheduler can greatly help to increase performance.

Finally, we plan to consider different program inputs and analyze their influence on predictive scheduling and run-time adaptation. We plan to use and extend techniques such as clustering [10], code versioning and program phase runtime adaptation [6,7] to improve the utilization and adaptation of all available resources in the future heterogeneous computing systems.

Acknowledgements

This work has been supported by the European Commission in the context of the SARC Integrated Project (EU contract 27648-FP6), the HiPEAC European Network of Excellence and the Ministry of Science and Technology of Spain and the European Union (FEDER) under contract TIN2007-60625.

References

1. Parboil benchmark suite, `http://www.crhc.uiuc.edu/impact/parboil.php`
2. CUDA Programming Guide 1.1. NVIDIA's website (2007)
3. Badia, R.M., Labarta, J., Sirvent, R., Pérez, J.M., Cela, J.M., Grima, R.: Programming grid applications with grid superscalar. J. Grid Comput. 1(2), 151–170 (2003)
4. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: Cellss: a programming model for the cell be architecture. In: SC 2006: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. ACM, New York (2006)
5. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE 93(2), 216–231 (2005); special issue on Program Generation, Optimization, and Platform Adaptation
6. Fursin, G., Cohen, A., O'Boyle, M., Temam, O.: A practical method for quickly evaluating program optimizations. In: Conte, T., Navarro, N., Hwu, W.-m.W., Valero, M., Ungerer, T. (eds.) HiPEAC 2005. LNCS, vol. 3793, pp. 29–46. Springer, Heidelberg (2005)
7. Fursin, G., Miranda, C., Pop, S., Cohen, A., Temam, O.: Practical run-time adaptation with procedure cloning to enable continuous collective compilation. In: Proceedings of the GCC Developers Summit (July 2007)

8. Gabb, H.A., Jackson, R.M., Sternberg, M.J.: Modelling protein docking using shape complementarity, electrostatics and biochemical information. Journal of Molecular Biology 272(1), 106–120 (1997)
9. Gelado, I., Kelm, J.H., Ryoo, S., Lumetta, S.S., Navarro, N., Hwu, W.m.W.: Cuba: an architecture for efficient cpu/co-processor data communication. In: ICS 2008: Proceedings of the 22nd annual international conference on Supercomputing, pp. 299–308. ACM, New York (2008)
10. Mackay, D.J.C.: Information Theory, Inference & Learning Algorithms. Cambridge University Press, Cambridge (2002)
11. Maheswaran, M., Siegel, H.J.: A dynamic matching and scheduling algorithm for heterogeneous computing systems. In: HCW 1998: Proceedings of the Seventh Heterogeneous Computing Workshop, Washington, DC, USA, p. 57. IEEE Computer Society, Los Alamitos (1998)
12. Oh, H., Ha, S.: A static scheduling heuristic for heterogeneous processors. In: Fraigniaud, P., Mignotte, A., Robert, Y., Bougé, L. (eds.) Euro-Par 1996. LNCS, vol. 1124, pp. 573–577. Springer, Heidelberg (1996)
13. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.m.W.: Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 73–82. ACM, New York (2008)
14. Sih, G.C., Lee, E.A.: A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. IEEE Trans. Parallel Distrib. Syst. 4(2), 175–187 (1993)
15. Stone, H.S.: Multiprocessor scheduling with the aid of network flow algorithms. IEEE Transactions on Software Engineering SE-3(1), 85–93 (1977)
16. Stratton, J., Stone, S., Hwu, W.m.: Mcuda: An efficient implementation of cuda kernels on multi-cores. Technical Report IMPACT-08-01, University of Illinois at Urbana-Champaign (March 2008)
17. Tanenbaum, A.S.: Modern Operating Systems. Prentice Hall PTR, Upper Saddle River (2001)
18. Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (2006)
19. Topcuoglu, H., Hariri, S., Wu, M.-Y.: Task scheduling algorithms for heterogeneous processors. In: Heterogeneous Computing Workshop, 1999 (HCW 1999) Proceedings. Eighth, pp. 3–14 (1999)
20. Topcuoglu, H., Hariri, S., Wu, M.-Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems 13(3), 260–274 (2002)
21. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing 27(1–2), 3–35 (2001)

Collective Optimization

Grigori Fursin and Olivier Temam

HiPEAC members
ALCHEMY Group, INRIA Futurs, France
{grigori.fursin,olivier.temam}@inria.fr

Abstract. Iterative compilation is an efficient approach to optimize programs on rapidly evolving hardware, but it is still only scarcely used in practice due to a necessity to gather a large number of runs often with the same data set and on the same environment in order to test many different optimizations and to select the most appropriate ones. Naturally, in many cases, users cannot afford a training phase, will run each data set once, develop new programs which are not yet known, and may regularly change the environment the programs are run on.

In this article, we propose to overcome that practical obstacle using *Collective Optimization*, where the task of optimizing a program leverages the experience of many other users, rather than being performed in isolation, and often redundantly, by each user. Collective optimization is an unobtrusive approach, where performance information obtained after each run is sent back to a central database, which is then queried for optimizations suggestions, and the program is then recompiled accordingly. We show that it is possible to learn across data sets, programs and architectures in non-dynamic environments using static function cloning and run-time adaptation without even a reference run to compute speedups over the baseline optimization. We also show that it is possible to simultaneously learn and improve performance, since there are no longer two separate training and test phases, as in most studies. We demonstrate that extensively relying on *competition* among pairs of optimizations (*program reaction to optimizations*) provides a robust and efficient method for capturing the impact of optimizations, and for reusing this knowledge across data sets, programs and environments. We implemented our approach in GCC and will publicly disseminate it in the near future.

1 Introduction

Many recent research efforts have shown how iterative compilation can outperform static compiler optimizations and quickly adapt to complex processor architectures [33,9,6,24,16,10,20,31,27,26,18,19]. Over the years, the approach has been perfected with fast optimization space search techniques, sophisticated machine-learning algorithms and continuous optimization [25,29,28,34,3,8,32,23,21]. And, even though these different research works have demonstrated significant performance improvements, the technique is far from mainstream in production environments. Besides the usual inertia for adopting new approaches, there are hard technical hurdles which hinder the adoption of iterative approaches.

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 34–49, 2009.

The most important hurdle is that iterative techniques almost all rely on a large number of training runs (either from the target program or other training programs) to *learn* the best candidate optimizations. And most of the aforementioned articles run the *same programs*, generated with the exact *same compiler* on the *same architecture* with the *same data sets*, and do this *a large number of times* (tens, hundreds or thousands of times) in order to deduce the shape of the optimization space. Naturally, in practice, a user is not going to run the same data set multiple times, will change architectures every so often, and will upgrade its compiler as well. We believe this practical issue of collecting a large number of training information, relying only on *production* runs (as opposed to training runs where results are not used) to achieve good performance is the crux of the slow adoption of iterative techniques in real environments.

We propose to address this issue with the notion of *Collective Optimization*. The principle is to consider that the task of optimizing a program is not an isolated task performed by each user separately, but a *collective* task where users can mutually benefit from the experience of others. Collective optimization makes sense because most of the programs we use daily are run by many other users, either globally if they are general tools, or within our or a few institutions if they are more domain-specific tools. Achieving collective optimization requires to solve both an *engineering* and a *research* issue.

The engineering issue is that users should be able to seamlessly share the outcome of their runs with other users, without impeding execution or compilation speed, or complicating compiler usage. The key research issue is that we must progressively improve overall program performance while, *at the same time*, we learn *how it reacts to the various optimizations*, all solely using *production runs*; in reality, there is no longer such a thing as a training phase followed by a test/use phase, both occur simultaneously. Moreover, we must understand *whether* it is possible and *how* to learn across data sets, programs or platforms. An associated research issue is to come up with a knowledge representation that is relevant across data sets, programs and platforms. Finally, because a user will generally run a data set only once, we must learn the impact of optimizations on program performance without even a *reference* run to decide whether selected optimizations improved or degraded performance compared to the baseline optimization.

In this article, we show that it is possible to continuously learn across data sets, programs or platforms, relying solely on production runs, and progressively improving overall performance across runs, reaching close to the best possible iterative optimization performance, itself achieved under idealized (and non-realistic) conditions. We show that extensively relying on *competition* among pairs of optimizations provides a robust and efficient method for capturing the impact of optimizations on program performance, without requiring reference runs and while remaining relevant across data sets, programs and architectures. While most recent research studies are focused on learning across programs [28,3,7], we find that, in practice, learning across data sets, and to a lesser extent, across architectures, is significantly more important and useful. Finally, we present a solution to the engineering issue in the form of an extension to GCC which

relies on a central database for transparently aggregating runs results from many users, and performing competitions between optimizations during runs.

2 Experimental Setup

In order to perform a realistic evaluation of collective optimization, each benchmark has to come with several data sets in order to emulate truly distinct runs. To our knowledge, only the MiDataSets [13] data set suite based on the MiBench [17] benchmark suite currently provides 20+ data sets for 26 benchmarks.

All programs are optimized using the GCC 4.2.0 compiler. The collective optimization approach and framework are compatible with other compilers, but GCC is now becoming a competitive optimization compiler with a large number of embedded program transformation techniques. We identified 88 program transformations, called through corresponding optimization flags, that are known to influence performance, and 8 parameters for each parametric optimization. These transformations are randomly selected to produce an optimization combination.

In order to unobtrusively collect information on a program run, and reoptimize the program, GCC is modified so as to add to each program a routine which is executed when the program terminates. This termination routine collects several information about the program (a program identifier, architecture and compiler identifiers, which optimizations were applied) and about the last run (performance measurements; currently, execution time and profiling information), and stores them into a remote database.

Then, it queries a server associated with the remote database in order to select the next optimizations combination. The recompilation takes place periodically (set by user) in the background, between two runs.[1] No other modification takes place until the next run, where the process loops again, as shown in Figure 1.

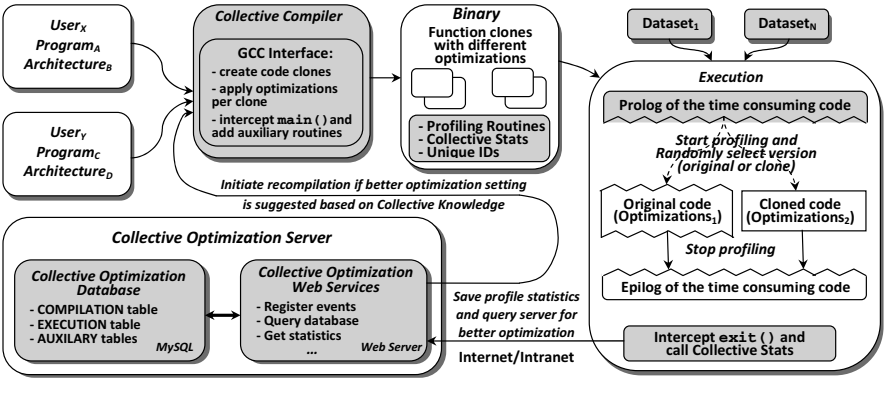


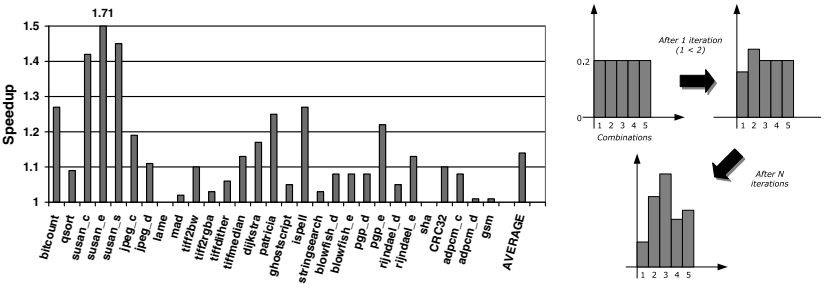
Fig. 1. Collective Optimization Framework

[1] Note that if the recompilation is not completed before another run starts, this latter run just uses the same optimizations as the previous run, and the evaluation of the new optimizations is just slightly delayed by one or a few runs.

The programs were compiled and run on three architectures - AMD Athlon XP 2800+ (AMD32) - 5 machines, AMD Athlon 64 3700+ (AMD64) - 16 machines and Intel Xeon 2.80GHz (IA32) - 2 machines.

3 Motivation

The experimental methodology of research in iterative optimization usually consists in running many times the same program on the same data set and on the same platform. Hence, it can be interpreted as an idealized case of collective optimization, where the experience of others (program, data set, platform) would always perfectly match the target run. In other words, it is a case where no experimental noise would be introduced by differences in data sets, programs or platforms. Consequently, iterative optimization can be considered as a performance upper-bound of collective optimization. Figure 2(a) shows the best speedup achieved for each benchmark and each data set (averaged over 20 distinct data sets) over the highest GCC optimization level (-O3) by selecting the best optimizations combination among 200 for each program and data set. This experiment implicitly shows that collective optimization has the potential to yield high speedups.



(a) (b)

Fig. 2. (a) Performance upper-bound of Collective Optimization (AMD Athlon 64 3700+, GCC 4.2.0), (b) Computing the probability distribution to select an optimization combination based on continuous competition between combinations

4 Overview

This section provides an overview of the proposed approach for collective optimization. The general principle is that performance data about each run is transparently collected and sent to a database; and, after each run, based on all the knowledge gathered so far, a new optimizations combination is selected and the program is recompiled accordingly. The key issue is which optimizations combination to select for each new run, in order to both gather new knowledge and keep improving average program performance as we learn.

In collective optimization, several global and program-specific probability distributions capture the accumulated knowledge. Combinations are randomly selected from one of several probability distributions which are progressively built at the remote server.

The different "maturation" stages of a program. For each program, and depending on the amount of accumulated knowledge, we distinguish three scenarios: (1) the server may not know the program at all (new program), (2) only have information about a few runs (infrequently used or a recently developed program), or (3) have information about many runs.

Stage 3: Program well known, heavily used. At that maturation stage, enough runs have been collected for that program that it does not need the experience of other programs to select the most appropriate optimizations combinations for itself. This knowledge takes the form of a program-specific probability distribution called d_3. Stage 3 corresponds to learning across data sets.

Stage 2: Program known, a few runs only. At that maturation stage, there is still insufficient information (program runs) to correctly predict the best combinations by itself, but there is already enough information to start "characterizing" the program behavior. This characterization is based on the comparison of the impact of optimizations combinations tried so far on the program against their impact on other programs (program reaction to optimizations). If two programs behave alike for a subset of combinations, they may well behave alike for all combinations. Based on this intuition, it is possible to find the best matching program, after applying a few combinations to the target program. Then, the target program probability distribution d_2 is given by the distribution d_3 of the matching program. This matching can be revisited with each additional information (run) collected for the target program. Stage 2 corresponds to learning across programs.

Stage 1: Program unknown. At that stage, almost no run has been performed, so we leverage and apply optimizations suggested by the "general" experience collected over all well-known programs. The resulting d_1 probability distribution is the unweighted average of all d_3 distributions of programs which have reached Stage 3. Stage 1 is an elementary form of learning across programs.

Selecting stages. A program does not follow a monotonic process from Stage 1 to Stage 3, even though it should intuitively mature from Stage 1 to Stage 2 to Stage 3 in most cases. There is a permanent *competition* between the different stages distributions (d_1, d_2, d_3). At any time, a program may elect to draw optimizations combinations from any stage distribution, depending on which one appears to perform best so far. In practice and on average, we find that Stage 3 (learning across data sets) is by far the most useful stage. Stage 1 and Stage 2 stages are respectively useful in the first ten, and the first hundreds runs of a program on average, but Stage 3 rapidly becomes dominant. The competition between stages is implemented through a "meta" distribution d_m, which reflects the current score of each stage distribution for a given program. Each new run is

a two-step random process: first, the server randomly selects the distribution to be used, and then, it randomly selects the combination using that distribution. How scores are computed is explained in Section 5. Using that meta-distribution, the distribution with the best score is usually favored.

5 Collective Learning

In this section, we explain in more detail how to compute the aforementioned distributions to achieve collective learning.

5.1 Building the Program Distribution d_3 Using Statistical Comparison of Optimizations Combinations

Comparing two combinations C_1, C_2. In order to build the aforementioned distributions, one must be able to compare the impact of any two optimizations combinations C_1, C_2 on program performance.

However, even the simple task of deciding whether $C_1 > C_2$ can become complex in a real context. Since the collective optimization process only relies on production runs, two runs usually correspond to two distinct data sets. Therefore, if two runs with respective execution times T_1 and T_2, and where optimizations combinations C_1 and C_2 have been respectively applied, are such that $T_1 < T_2$, it is not possible to deduce that $C_1 > C_2$.

To circumvent that issue, we perform run-time comparison of *two* optimizations combinations using *cloned* functions. C_1 and C_2 are respectively applied to the clones f_1 and f_2 of a function f. At run-time, for each call to f, either f_1 or f_2 is called; the clone called is randomly selected using an additional branch instruction and a simple low-overhead pseudo-random number generation technique emulating uniform distribution. Even if the routine workload varies upon each call, the random selection of the clone to be executed ensures that the average workload performed by each clone is similar. As a result, the non-optimized versions of f_1 and f_2 account for about the same fraction of the overall execution time of f. Therefore, if the average execution time of the clone optimized with C_1 is smaller than the average execution time of the clone optimized with C_2, it is often correct to deduce that C_1 is better than C_2, i.e., $C_1 > C_2$. This statistical comparison of optimizations combinations requires no reference, test or training run, and the overhead is negligible.

We have shown in [14] the possibility to detect the influence of optimizations for statically compiled programs with stable behavior using function cloning and run-time low-overhead phase detection. Stephenson et al. [30] and Lau et al. [22] demonstrated how to evaluate different optimizations for programs with irregular behavior in dynamic environments using random function invocations and averaging collected time samples across a period of time. We combined these techniques to enable run-time transparent performance evaluation for statically-compiled programs with any behavior here.

On the first program run, profiling information is collected and sent to the database. All the most time-consuming routines accounting for 75% or more of

the total execution time and with an average execution time per call greater than a platform-specific threshold are cloned. The purpose of this threshold is to ensure that the overhead of the additional branch instruction remains negligible (less than 0.1%) compared to the average execution time per function call. Since profiling information is periodically collected at random runs, more routines can be added during the next runs (for instance if different parts of the call graph are reached depending on the data sets), the target routines are not set in stone after the first run. More implementation details are provided in Section 6.

Computing d_3. When two combinations C_1 and C_2 are compared on a program using the aforementioned cloned routines, the only information recorded is whether $C_1 > C_2$ or $C_1 < C_2$. Implicitly, a run is a *competition* between two optimizations combinations, and the winning combination scores 1 and the losing is 0 as shown in Figure 2(b). These scores are cumulated for each combination and program. The scores are then normalized per combination, by the number of times the combination was tried (thus implicitly decreasing the average score of the losing combination). Then the overall distribution is normalized so that the sum of all combinations scores (probabilities) is 1.

Because this distribution only reflects the *relative* "merit" of each combination, and not the absolute performance (e.g., execution time or speedup), it is a fairly resilient metric, tolerant to variations in measurements.

5.2 Building the Aggregate Distribution d_1

d_1 is simply the average of all d_3 distributions of each program. d_1 reflects the most common cases: which optimizations combinations perform best in general. It is also possible to compose more restricted aggregate distributions, such as per architecture, per compiler, per programs subsets,..., though this is left for future work.

5.3 Building the Matching Distribution d_2

Stage 2 is based on the intuition that it is unlikely that all programs exhibit widely different behavior with respect to compiler optimizations, or conversely that, once the database is populated with a sufficient number of programs, it is likely that a new program may favor some of the same optimizations combinations as some of the programs already in the database. The main difficulty is then to identify which programs best correspond to the current target one. Therefore, we must somehow *characterize* programs, and this characterization should reflect which optimizations combinations a program favors.

As for d_3, we use the metric-independent comparison between two optimizations combinations C_1 and C_2. E.g., $C_1 > C_2$ is a *reaction to program optimizations* and is used as one *characterization* of the program. Let us assume that $C_1 > C_2$ for the target program P and $C_1 > C_2$ for a program P' and $C_1 < C_2$ for a program P'' compared against P. Then, P' gets a score of 1, and P'' a score of 0. The program with the best score is considered the *matching* program, and d_2 is set to the d_3 of that program. In other words, for d_2 we use a competition

among *programs*. The more combinations pairs (reactions to optimizations) are compared, the more accurate and reliable the program matching.

Still, we observed that beyond 100 characterizing combinations pairs (out of $C^2_{100} = \frac{200\times199}{/}2 = 19900$ possible combinations pairs), performance barely improves. In addition, it would not be practical to recompute the matching upon each run based on an indefinitely growing number of characterizations. Therefore, we restrict the characterization to 100 combinations pairs, which are collected within a rolling window (FIFO). However, the window only contains distinct optimizations combinations pairs. The rolling property ensures that the characterization is permanently revisited and rapidly adapted if necessary. The matching is attempted as soon as one characterization is available in the window, and continuously revisited with each new modification of the rolling window.

Cavazos et al. [7] have shown that it is possible to improve similar program characterizations by identifying and then restricting to optimizations which carry the most information using the *mutual information* criterion. However, these optimizations do not necessarily perform best, they are the most *discriminatory* and one may not afford to "test" them in production runs. Moreover, we will later see that this approach could only yield marginal improvement in the start-up phase due to the rapid convergence of Stage $3/d_3$.

5.4 Scoring Distributions

As mentioned in Section 4, a meta-distribution is used to select which stage distribution is used to generate the next optimizations combination. For each run, two distributions d and d' are selected using two draws from the meta-distribution (they can be the same distributions). Then, an optimizations combination is drawn from each distribution (C_1 using d and C_2 using d'), which will compete during the run. Scoring is performed upon the run completion; note that if C_1 and C_2 are the same combinations, no scoring takes place.

Let us assume, for instance, that for the run, $C_1 > C_2$. If, according to d, $C_1 > C_2$ also, then one can consider that d "predicted" the result right, and gets a score of 1. Conversely, it would get a score of 0. The server also keeps track of the number of times each distribution is drawn, and the distribution value in the meta-distribution is the ratio of the sum of all its scores so far and the number of times it was drawn. Implicitly, its score decreases when it gets a 0, increases when it gets a 1, as for individual distributions.

This scoring mechanism is robust. If a distribution has a high score, but starts to behave poorly because the typical behavior of the program has changed (e.g., a very different kind of data sets is used), then its score will plummet, and the relative score of other distributions will comparatively increase, allowing to discover new strong combinations. Note that d_3 is updated upon every run (with distinct combinations), even if it was not drawn, ensuring that it converges as fast as possible.

6 Collective Compiler

Program identification. At the moment, a program is uniquely identified using a 32-byte MD5 checksum of all the files in its source directory. This identifier is sufficient to distinguish most programs, and it has the added benefit of not breaking confidentiality: no usable program information is sent to the database. In the future, we also plan to use the vector of program reactions to transformations as a simple and practical way to characterize programs based solely on execution time.

Termination routine. In order to transparently collect run information, we modified GCC to intercept the compilation of the `main()` function, and to insert another interceptor on the `exit()` function. Whenever the program execution finishes, our interceptor is invoked and it in turn checks whether the *Collective Stats Handler* exists, invokes it to send program and run information to the *Collective Optimization Database*. At any time, the user can opt in or out of collective optimization by setting or resetting an environment variable.

Cloning. As mentioned before, optimizations combinations are evaluated through cloned routines. These routines are the most time-consuming program routines (top 3 routines and/or 75% or more of the execution time). They are selected using the standard `gprof` utility. The program is profiled at the first and then random runs. Therefore, the definition of the top routines can change across runs. We progressively build an average ranking of the program routines, possibly learning new routines as they are exercised by different data sets. The speedups mentioned in the following performance evaluation section are provided with respect to a *non-instrumented* version of the program, i.e., they factor in the instrumentation, which usually has a negligible impact.

We modified GCC to enable function cloning and be able to apply different optimizations directly to clones and original functions. This required changes in the core of the compiler since we had to implement full replication of parts of a program AST, and to change the optimization pass manager to be able to select specific optimizations combinations on a function level. When GCC clones a function, it inserts profiling calls at the prolog and epilog of the function, replaces `static` variables and inserts additional instructions to randomly select either the original or the cloned version.

Security. The concept of collective optimization raises new issues, especially security. First, very little program information is in fact sent to the database. The profile routine names are locally hashed using MD5, and only run-time statistics (execution times) are sent. Second, while we intend to set up a global and openly accessible database, we do expect companies will want to set up their own internal collective optimization database. Note that they can then get the best of both worlds: leverage/read information accessible from the global database, while solely writing their runs information to their private database. At the moment, our framework is designed for a single database, but this two-database system is a rather simple evolution.

7 Performance Evaluation

In Figure 3(a), *Collective* corresponds to the full process described in earlier sections, where the appropriate distribution is selected using the meta-distribution before every run; performance is averaged over all programs (for instance, `Run=1` corresponds to performance averaged over 1 random run for each program). For each program, we have collected 20 data sets and can apply 200 different optimizations combinations, for a total of 4000 distinct runs per program. The main approximation of our evaluation lays in the number of data sets; upon each run, we (uniformly) randomly select one among 20 data sets. However, several studies have shown that data sets are often clustered within a few groups breeding similar behavior [11], so that 20 data sets exhibiting sufficiently distinct behavior, may be considered a non-perfect but reasonable emulation of varying program behavior across data sets. In order to further assess the impact of using a restricted number of data sets, we have evaluated the extreme case where a single data set is used. These results are reported in Figure 3(b), see *Single data set*, where a single data set is used per program in each experiment, and then, for each x-axis value (number of runs), performance is averaged over all programs and all data sets. Using a single data set improves convergence speed though only moderately, suggesting *Collective* could be a slightly optimistic but reasonable approximation of a real case where all data sets are distinct.

After 10000 runs per program, the average *Collective* speedup, 1.11, is fairly close to the *Best* possible speedup, 1.13, the asymptotic behavior of single-data set experiments. The other graphs (*d1, d2, d3*) report the evolution of the average performance of optimizations combinations drawn from each distribution. At the bottom of the figure, the grey filled curve corresponds to the meta score of d_3, the black one to d_2 and the white one to d_1.

Learning across programs. While the behavior of d_2 in Figure 3(a) suggests that learning across programs yields modest performance improvements, this experiment is partly misleading. d_3 rapidly becomes a dominant distribution, and as explained above, quickly converges to one or a few top combinations due to restricted interval polling. d_2 performance will improve as more characterizing optimizations combinations pairs fill up the rolling window. And Figure 3(b) shows that without d_2, the meta distribution does not converge as fast or to an as high asymptotic value.

Collective *versus* d_3. While the better performance of d_3 over *Collective*, in Figure 3(a), suggests this distribution should solely be used, one can note its performance is not necessarily the best in the first few runs, which is important for infrequently used codes. Moreover, the average *Collective* performance across runs becomes in fact very similar after d_3 has become the dominant distribution, since mostly d_3 combinations are then drawn. But a more compelling reason for privileging *Collective* over d_3 is the greater robustness of collective optimization thanks to its meta-distribution scheme.

In Figure 3(b), we have tested collective optimization without either d_1, d_2 or neither one. In the latter case, we use the uniform random distribution to

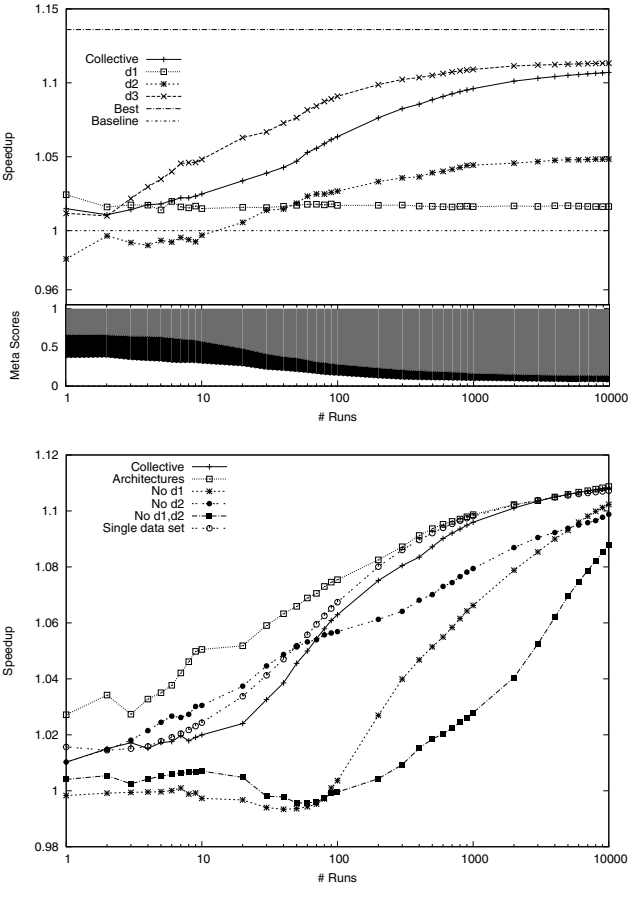


Fig. 3. (a) Average performance of collective optimization and individual distributions (bottom: meta-scores of individual distributions; grey is d_3, black is d_2, white is d_1), (b) Several collective optimization variants

discover new optimizations, and the meta-distribution arbitrates between d_3 and *uniform*; by setting the uniform distribution initial meta-score to a low value with respect to d_3, we can both quickly discover good optimizations without degrading average performance;[2] the uniform distribution is not used when only d_1 or d_2 are removed. As shown in Figure 3(b), collective optimization converges more slowly when either d_1, d_2 or both are not used. These distributions help in two ways. d_1 plays its main role at start-up, by bringing a modest average 2% improvement, and performance starts lower when it is not used. Conversely, d_2 is not useful at start-up, but provides a performance boost after about 50 to 100 runs when its window is filled and the matching is more accurate. d_2 does in fact significantly help improve the performance of *Collective* after 100 runs, but

[2] This is important since the average speedup of the uniform distribution alone is 0.7.

essentially by discovering good new optimizations, later adopted by d_3, rather than due to the intrinsic average performance of the optimizations combinations suggested by d_2.

Learning across architectures. Besides learning across data sets and programs, we have also experimented with learning across architectures. We have collected similar runs on an Athlon 32-bit (AMD32) architecture and an Intel 32-bit (IA32) architecture (recall all experiments above are performed on an Athlon 64-bit architecture), and we have built the d_3 distributions for each program. At start-up time, on the 64-bit architecture, we now use a d_4 distribution corresponding to the d_3 distribution for this program but other architectures (and 19 data sets, excluding the target data set); the importance of d_4 will again be determined by its meta-score. The rest of the process remains identical. The results are reported in curve *Architectures* on Figure 3(b). Start-up performance does benefit from the experience collected on the other architectures. However, this advantage fades away after about 2000 runs. We have also experimented with simply initializing d_3 with the aforementioned d_4 instead of using a separate d_4 distribution. However the results were poorer because the knowledge acquired from other architectures was slowing down the rate at which d_3 could learn the behavior of the program on the new architecture.

8 Background and Related Work

Iterative or adaptive compilation techniques usually attempt to find the best possible combinations and settings of optimizations by scanning the space of all possible optimizations. [33,9,6,24,10,20,31,27,26,19] demonstrated that optimizations search techniques can effectively improve performance of statically compiled programs on rapidly evolving architectures, thereby outperforming state-of-the-art compilers, albeit at the cost of a large number of exploration runs.

Several research works have shown how machine-learning and statistical techniques [25,29,28,34] can be used to select or tune program transformations based on program features. Agakov et al. [3] and Cavazos et al. [8] use machine-learning to focus iterative search using either syntactic program features or dynamic hardware counters and multiple program transformations. Most of these works also require a large number of training runs. Stephenson et al. [28] show more complementarity with collective optimization as program matching is solely based on static features.

Several frameworks have been proposed for continuous program optimization [4,32,23,21]. Such frameworks tune programs either during execution or offline, trying different program transformations. Such recent frameworks like [21] and [23] pioneer lifelong program optimization, but they expose the concept rather than research practical knowledge management and selection strategies across runs, or unobtrusive optimization evaluation techniques. Several recent research efforts [14,22,30] suggest to use procedure cloning to search for best optimizations at run-time. In this article we combine and extend techniques from [14] that are compatible with regular scientific programs and use low-overhead

run-time phase detection, and methods from [30,22] that can be applied to programs with irregular behavior in dynamic environments by randomly executing code versions and using statistical analysis of the collected execution times with a confidence metric. Another recent research project investigates the potential of optimizing static programs across multiple data sets [13] and suggests this task is tractable though not necessarily straightforward.

The works closest to ours are by Arnold et al. [5] and Stephenson [30]. The system in [5] collects profile information across multiple runs of a program in IBM J9 Java VM to selectively apply optimizations and improve further invocations of a given program. However it doesn't enable optimization knowledge reuse from different users, programs and architectures. On the contrary, Stephenson tunes a Java JIT compiler across executions by multiple users. While several aspects of his approach is applicable to static compilers, much of his work focuses on Java specifics, such as canceling performance noise due to methods recompilation, or the impact of garbage collection. Another distinctive issue is that, in a JIT, the time to predict optimizations and to recompile must be factored in, while our framework tolerates well long lapses between recompilations, including several runs with the same optimizations. Finally, we focus more on the impact of data sets from multiple users and the optimization selection robustness (through competitions and meta-distribution).

9 Conclusions and Future Work

The first contribution of this article is to identify the true limitations of the adoption of iterative optimization in production environments, while most studies keep focusing on showing the performance potential of iterative optimization. We believe the key limitation is the large amount of knowledge (runs) that must be accumulated to efficiently guide the selection of compiler optimizations. The second contribution is to show that it is possible to simultaneously *learn* and *improve* performance across runs. The third contribution is to propose multi-level competition (among optimizations and their distributions which capture different program knowledge maturation stages, and among programs) to understand the impact of optimizations without even a reference run for computing speedups, while ensuring optimization robustness. The program reactions to transformations used to build such distributions provide a simple and practical way to characterize programs based solely on execution time. The fourth contribution is to highlight that knowledge accumulated across data sets for a single program is more useful, in the real and practical context of collective optimization, than the knowledge accumulated across programs, while most iterative optimization studies focus on knowledge accumulated across programs; we also conclude that knowledge across architectures is useful at start-up but does not bring any particular advantage in steady-state performance. The fifth and final contribution is to address the engineering issue of unobtrusively collecting runs information for statically-compiled programs using function cloning and run-time adaptation mechanism.

The collective optimization approach opens up many possibilities which can be explored in the future. We plan to use it to automatically and continuously tune default GCC optimization heuristic or individual programs using recent plugin system for GCC [15,2] that allows to invoke transformations directly, change their parameters, orders per function or even add plugins with new transformations to improve performance, code size, power, and so on. After sufficient knowledge has been accumulated, the central database may become a powerful tool for defining truly representative benchmarks. We can also refine optimizations at the data set level by clustering data sets and using our cloning and run-time adaptation techniques to select the most appropriate optimizations combinations or even reconfigure processor at run-time thus creating self-tuning intelligent ecosystem. We plan to publicly disseminate our collective optimization framework, the run-time adaptation routines for GCC based on [12,8] as well as the associated central database at [1] in the near future.

Acknowledgments

This research was partially supported by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC), and the IST STREP project MILEPOST. The authors would also like to thank colleagues and anonymous reviewers for their feedback.

References

1. Continuous Collective Compilation Framework, `http://cccpf.sourceforge.net`
2. GCC Interactive Compilation Interface, `http://gcc-ici.sourceforge.net`
3. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M., Thomson, J., Toussaint, M., Williams, C.: Using machine learning to focus iterative optimization. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO) (2006)
4. Anderson, J., Berc, L., Dean, J., Ghemawat, S., Henzinger, M., Leung, S., Sites, D., Vandevoorde, M., Waldspurger, C., Weihl, W.: Continuous profiling: Where have all the cycles gone. In: Technical Report 1997-016. Digital Equipment Corporation Systems Research Center, Palo Alto, CA (1997)
5. Arnold, M., Welc, A., Rajan, V.T.: Improving virtual machine performance using a cross-run profile repository. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005) (2005)
6. Bodin, F., Kisuki, T., Knijnenburg, P., O'Boyle, M., Rohou, E.: Iterative compilation in a non-linear optimisation space. In: Proceedings of the Workshop on Profile and Feedback Directed Compilation (1998)
7. Cavazos, J., Dubach, C., Agakov, F., Bonilla, E., O'Boyle, M., Fursin, G., Temam, O.: Automatic performance model construction for the fast software exploration of new hardware designs. In: Proceedings of the International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES) (October 2006)

8. Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M., Temam, O.: Rapidly selecting good compiler optimizations using performance counters. In: Proceedings of the 5th Annual International Symposium on Code Generation and Optimization (CGO) (March 2007)
9. Cooper, K., Schielke, P., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp. 1–9 (1999)
10. Cooper, K., Subramanian, D., Torczon, L.: Adaptive optimizing compilers for the 21st century. Journal of Supercomputing 23(1) (2002)
11. Eeckhout, L., Vandierendonck, H., Bosschere, K.D.: Quantifying the impact of input data sets on program behavior and its applications. Journal of Instruction-Level Parallelism 5, 1–33 (2003)
12. Franke, B., O'Boyle, M., Thomson, J., Fursin, G.: Probabilistic source-level optimisation of embedded programs. In: Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES) (2005)
13. Fursin, G., Cavazos, J., O'Boyle, M., Temam, O.: Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. In: De Bosschere, K., Kaeli, D., Stenström, P., Whalley, D., Ungerer, T. (eds.) HiPEAC 2007. LNCS, vol. 4367, pp. 245–260. Springer, Heidelberg (2007)
14. Fursin, G., Cohen, A., O'Boyle, M., Temam, O.: A practical method for quickly evaluating program optimizations. In: Conte, T., Navarro, N., Hwu, W.-m.W., Valero, M., Ungerer, T. (eds.) HiPEAC 2005. LNCS, vol. 3793, pp. 29–46. Springer, Heidelberg (2005)
15. Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., O'Boyle, M.: Milepost gcc: machine learning based research compiler. In: Proceedings of the GCC Developers Summit (June 2008)
16. Fursin, G., O'Boyle, M., Knijnenburg, P.: Evaluating iterative compilation. In: Pugh, B., Tseng, C.-W. (eds.) LCPC 2002. LNCS, vol. 2481, pp. 305–315. Springer, Heidelberg (2005)
17. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: Proceedings of the IEEE 4th Annual Workshop on Workload Characterization, Austin, TX (December 2001)
18. Heydemann, K., Bodin, F.: Iterative compilation for two antagonistic criteria: Application to code size and performance. In: Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems, colocated with CGO (2006)
19. Hoste, K., Eeckhout, L.: Cole: Compiler optimization level exploration. In: Proceedings of International Symposium on Code Generation and Optimization (CGO) (2008)
20. Kulkarni, P., Zhao, W., Moon, H., Cho, K., Whalley, D., Davidson, J., Bailey, M., Paek, Y., Gallivan, K.: Finding effective optimization phase sequences. In: Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES), pp. 12–23 (2003)
21. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO), Palo Alto, California (March 2004)
22. Lau, J., Arnold, M., Hind, M., Calder, B.: Online performance auditing: Using hot optimizations without getting burned. In: Proceedings of the ACM SIGPLAN Conference on Programming Languaged Design and Implementation (PLDI) (2006)

23. Lu, J., Chen, H., Yew, P.-C., Hsu, W.-C.: Design and implementation of a lightweight dynamic optimization system. Journal of Instruction-Level Parallelism 6 (2004)
24. Matteo, F., Johnson, S.: FFTW: An adaptive software architecture for the FFT. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, Seattle, WA, vol. 3, pp. 1381–1384 (May 1998)
25. Monsifrot, A., Bodin, F., Quiniou, R.: A machine learning approach to automatic production of compiler heuristics. In: Scott, D. (ed.) AIMSA 2002. LNCS, vol. 2443, pp. 41–50. Springer, Heidelberg (2002)
26. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), pp. 319–332 (2006)
27. Singer, B., Veloso, M.: Learning to predict performance from formula modeling and training data. In: Proceedings of the Conference on Machine Learning (2000)
28. Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). IEEE Computer Society, Los Alamitos (2005)
29. Stephenson, M., Martin, M., O'Reilly, U.: Meta optimization: Improving compiler heuristics with machine learning. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 77–90 (2003)
30. Stephenson, M.W.: Automating the Construction of Compiler Heuristics Using Machine Learning. PhD thesis, MIT, USA (2006)
31. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.: Compiler optimization-space exploration. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), pp. 204–215 (2003)
32. Voss, M., Eigenmann, R.: Adapt: Automated de-coupled adaptive program transformation. In: Proceedings of the International Conference on Parallel Processing (ICPP) (2000)
33. Whaley, R., Dongarra, J.: Automatically tuned linear algebra software. In: Proceedings of the Conference on High Performance Networking and Computing (1998)
34. Zhao, M., Childers, B.R., Soffa, M.L.: A model-based framework: an approach for profit-driven optimization. In: Proceedings of the Interational Conference on Code Generation and Optimization (CGO), pp. 317–327 (2005)

High Speed CPU Simulation Using LTU Dynamic Binary Translation*

Daniel Jones and Nigel Topham

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh
Great Britain
daniel.jones@ed.ac.uk, npt@inf.ed.ac.uk

Abstract. In order to increase the speed of dynamic binary translation based simulators we consider the translation of large translation units consisting of multiple blocks. In contrast to other simulators, which translate hot blocks or pages, the techniques presented in this paper profile the target program's execution path at runtime. The identification of hot paths ensures that only executed code is translated whilst at the same time offering greater scope for optimization. Mean performance figures for the functional simulation of EEMBC benchmarks show the new simulation techniques to be at least 63% faster than basic block based dynamic binary translation.

1 Introduction

Simulators play an important role in the design of today's high performance microprocessors. They support design-space exploration, where processor characteristics such as speed and power consumption are accurately predicted for different architectural models. The information gathered enables designers to select the most efficient processor designs for fabrication. Simulators also provide a platform on which experimental ISAs can be tested, and new compilers and applications may be developed and verified. They help to reduce the overall development time for new microprocessors by allowing concurrent engineering during the design phase. This is especially important for embedded system-on-chip designs, where processors may be extended to support specific applications.

The ever-increasing complexity of modern microprocessors exacerbates the central challenge of achieving high-speed simulation whilst at the same time retaining absolute modelling accuracy. Simulators must model the different processor units which perform sophisticated functions and emulate events such as interrupts and exceptions. Many embedded processors now incorporate memory management hardware to support multi-tasking operating systems.

This paper is concerned with that class of simulator which provides *accurate* and *observable* modelling of the entire processor state. This is possible to achieve

* Research supported by EPSRC under grant EP/D50399X/1.

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 50–64, 2009.

by operating at the register transfer level, but such simulators are very slow. In contrast, compiled simulation, which can be many orders of magnitude faster, does not have the same degree of observability and can only be used in situations where the application code is known in advance and is available in source form. Programs which require an operating system or which are shrink-wrapped can not benefit from compiled simulation.

Dynamic Binary Translation (DBT) on the other hand combines interpretive and compiled simulation techniques in order to maintain high speed, observability and flexibility. However, achieving accurate state observability remains in tension with high speed simulation. The Edinburgh High Speed (EHS) DBT simulator developed at the Institute for Computing Systems Architecture, Edinburgh University, was used to carry out this research.

Section 2 presents an overview of the EHS simulator and section 3 details the different types of Large Translation Unit (LTU). Section 4 explains the profiling methods used to identify the different translation-units, whilst section 5 describes the mechanics of DBT simulation. Performance results are presented in section 6, followed by related work in section 7 and the conclusions in section 8.

2 Simulator Overview

The Edinburgh High Speed simulator [13] presented in this paper is designed to be target-adaptable, with the initial target being the ARC 700 processor which implements the ARCompact instruction set architecture.

It can be run as either a user-level (emulated system calls) or system-level simulator. In full-system mode, the simulator implements the processor, its memory sub-system (including MMU), and sufficient interrupt-driven peripherals to simulate the boot-up and interactive operation of a complete Linux-based system.

The simulator possesses two simulation modes: an interpretive mode which provides precise observability after each instruction, and a DBT mode which provides similarly precise observability between successive basic blocks. By capturing all architecturally visible CPU state changes the simulator is able to support high speed hardware-software co-verification. The simulator's underlying system architecture for handling memory access, I/O, interrupts, and exceptions is the same for both interpreted and translated modes. This means that the simulator is able to switch, at basic block boundaries, between the two simulation modes at runtime.

In DBT simulation mode, target binaries can be simulated at speeds in excess of ten times faster than for interpretive mode. When operating in DBT mode the simulator initially interprets all instructions, discovering and profiling basic blocks as they are encountered. After interpreting a set number of blocks, the target program's execution path is analysed for frequently executed basic blocks, or LTUs, which are then marked for binary translation. After the hot blocks, or LTUs, have been dynamically translated and loaded, they are from that moment on simulated by calling the corresponding translations directly.

The simulation of a target executable may be interrupted within a translated block and restarted at the current program counter. This enables translated

blocks to raise exceptions part-way through, after which the remaining instructions in the block will be interpreted.

The simulator can also be directed to retain the translations produced from the simulation of a target executable. These translations can then be used by the next simulation run of the same executable. This mechanism enables a library of translations to be built up for a target program. The maximum simulation speed for an executable can therefore be ascertained once all of the target instructions have been translated by previous simulation runs.

3 Large Translation Units

Typically, in DBT simulators, the unit of translation is either the target instruction or the basic block. By increasing the size of the translation-unit it is possible to achieve significant speedups in simulation performance. Translation-units are the objects which are dynamically compiled and loaded as Translated Functions (TFs). The TFs are then called directly to emulate the target code at very high speed.

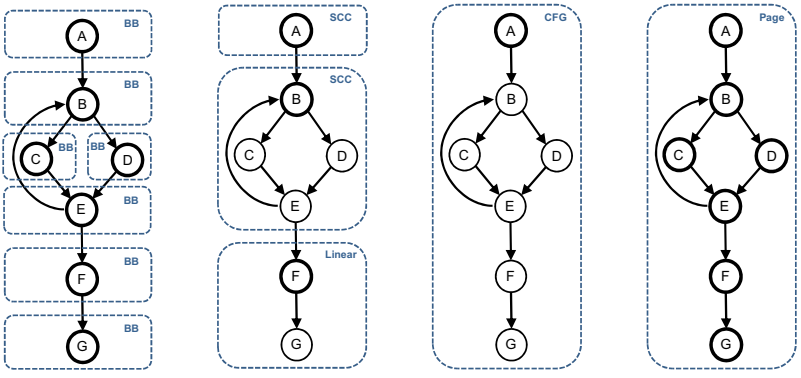
The increase in performance is due to two main factors. Firstly, LTUs offer the compiler greater scope for optimization across multiple blocks, rather than just a single basic block. Secondly, more time is spent simulating within a given TF and less time is spent returning to the main simulation loop.

In this paper we consider three different types of LTU, in addition to the basic block translation-unit. An LTU in this context is a group of basic blocks connected by control-flow arcs, which may have one or more entry and exit points. The different translation modes incorporated into the EHS simulator include Basic Block (BB), Strongly Connected Components (SCC), Control Flow Graph (CFG) and Physical Page (Page).

In contrast to other DBT simulators, the EHS simulator profiles the target program's execution in order to discover hot paths rather than to identify hot blocks or pages, areas of which may be infrequently executed. The target program is profiled and the translation-units created on a per physical-page basis. Figure 1 shows the different translation-unit types and and their entry points. Example target-program CFGs are shown divided into separate translation-units in accordance with DBT mode. The possible entry point/s into a translation-unit varies with DBT mode. However, entry is always to the address of the first instruction within a basic block.

In Basic Block DBT, those basic blocks which are frequently executed at simulation time are identified and then sent for binary translation. When the PC value subsequently matches a translated basic block's start address, instead of interpreting the block, the translated code associated with the basic block is called directly.

In SCC DBT, the block execution path is analysed to discover SCCs (strongly connected blocks) and linear block regions. Frequently executed SCCs (and linear regions) are then marked for translation. When the PC value subsequently matches a translated SCC's (or linear region's) root block address, the translated code associated with the SCC (or linear region) is called directly.



(a) BB (b) SCC (c) CFG (d) Page

Fig. 1. Different Types of Translation Unit. Each figure shows a target-program CFG divided into DBT mode translation-units. Dotted lines indicate the separate translation-units whilst thick-edged circles show the possible entry points.

In CFG DBT, the block execution path is analysed to discover the CFGs. Frequently executed CFGs are then marked for translation. When the PC value subsequently matches a translated CFG's root block address, the translated code associated with the CFG is called directly.

In Page DBT, the block execution path is analysed to discover the CFGs. All the CFGs found are then binary translated as a whole. When the PC value subsequently matches the start address of any block within a translated unit, the translated code associated with the block is called directly.

4 Dynamic Execution Profiling

Simulation time is partitioned into *epochs*, where each epoch is defined as the interval between two successive binary translations. During each epoch, new translation-units may be discovered; previously seen but not-translated translation-units may be re-interpreted; translated translation-units may be discarded (*e.g.* self-modifying code); and translated translation-units may be executed. Throughout this process the simulator continues profiling those basic blocks which are still being interpreted. The end of each simulation epoch is reached when the number of interpreted basic blocks is equal to a predefined number.

Execution profiles for each physical-page are built up during the simulation epoch. For BB DBT this involves simply maintaining a count of the number of times individual basic blocks have been interpreted. In LTU DBT (SCC, CFG or Page) a page-CFG is produced for each physical-page. An LTU profile is created by adding the next interpreted block to the page-CFG, as well as incrementing the block's execution count.

Figure 2 shows examples of the different types of page-CFG that may be created during simulation. A page-CFG can consist of a single CFG or multiple

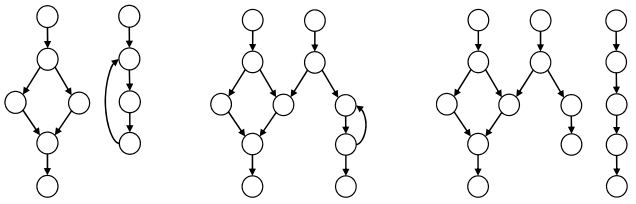


Fig. 2. Example page-CFG Configurations. A page-CFG may contain any number of (a) separate CFGs, (b) combined CFGs or (c) a mixture of both.

CFGs, in which case they may be separate, combined or a mixture of both types. In order that a page-CFG is not "broken" by an interrupt or an exception, the block sequences for the different processor interrupt states are tracked separately.

At the end of each simulation epoch the page-CFGs are analysed to retrieve the constituent translation-units. In the case of SCC DBT, Tarjan's algorithm [12] is applied to each CFG in order to extract the SCC translation-units. Regions of linear basic blocks are also identified as another translation-unit. In CFG DBT, the translation-units are extracted by tracing the CFG paths starting at each of the root nodes. No further processing is required for Page DBT as the translation-unit is the page-CFG itself.

5 Dynamic Binary Translation

The Dynamic Binary Translation simulation process is outlined in Fig. 3. The simulator looks up the next PC address in a fast Translation Cache (TC) which return translations. The TC, which is indexed by basic block start address, contains a pointer to the Translated Function (TF) for the corresponding translation-unit.

If the next PC address hits in the TC, its TF is called, thereby emulating directly the translation-unit. In addition to performing all of the operations within the translation-unit, all state information for the simulated processor, including the PC value, is updated prior to exiting the TF.

If the next PC address misses in the TC, it is looked up (physical address) in the Translation Map (TM). The TM contains an entry for every translated translation-unit. Each TM entry contains the translation-unit's entry address and a pointer to its TF which is used to call the corresponding host-code function. If the PC address hits in the TM the corresponding TF pointer is cached in the TC and the TF called.

If the PC address also misses in the TM, the translation-unit at this address has not yet been translated. The basic block must therefore be interpreted and the appropriate profiling information gathered. In the case of BB DBT, an entry for the basic block is cached in the Epoch Block Cache (EBC) to record all blocks interpreted during the current epoch. In the case of LTU DBT, the basic block is added to the Epoch CFG Cache (ECC) which builds up a page-CFG of

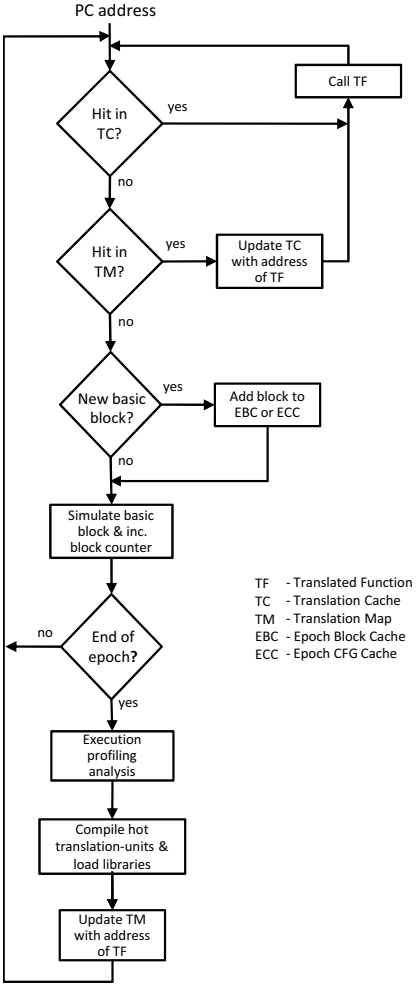


Fig. 3. Simulation Flow Chart

the target program's execution (interpreted) for the current epoch. Instances of the epoch cache exist for each physical page.

At the end of each simulation epoch, a profiling analysis phase is initiated prior to binary translation. In the case of BB DBT, the EBCs are scanned for frequently executed blocks. In SCC and CFG DBT, the page-CFGs cached in the ECCs are searched for frequently executed translation-units. No further analysis is required for Page DBT.

Those translation-units which have been interpreted at least as many times as the predefined translation threshold are marked for translation. The metric used to determine whether a translation-unit is hot depends on the translation-unit type. For BB translation it is the number of executions, for SCC and CFG translation it is the number of root node executions. Page translation-units are always translated.

The hot translation-units are then translated in batches comprising of translation units from the same physical page. The grouping of translation-units in this way enables all of the translations for a physical page to be easily discarded if for example, a target page is overwritten by self-modifying code or reused by a different process.

Each translation-unit is converted in to a C code function which emulates the target instructions. The functions are then compiled using GCC in to a shared library which is loaded by the dynamic linker. Finally, the TM is updated with the address of each newly translated TF. The next time one of the recently translated translation-units needs simulating, it will hit in the TM, an entry will be added to the TC and its TF called.

Each target instruction within a basic block is translated in to C code which emulates the instruction on the simulator. The processor state is updated by each instruction, with the exception of the PC which is updated at the end of the block, or on encountering an exception. An exception (MMU miss) causes an immediate return from the TF to the main simulation loop. Any pending interrupts are checked for at the end of each block before simulating the next block within the translation-unit. If a pending interrupt exists control is transferred back to the main simulation loop. All edges connecting basic blocks are recorded during profiling. This enables the program's control flow to be replicated within TFs with the use of `GOTO` statements. In Page DBT, the C code function representing a TF contains a jump table at the beginning so that simulation may commence at any block within the translation-unit.

6 Results

In this section we present the results from simulating a subset of the EEMBC benchmark suite on the Edinburgh High Speed simulator. The benchmarks were run on the simulator in order to compare the performance of the four DBT modes. All benchmarks were compiled for the ARC 700 architecture using `gcc` version 4.2.1 with `-O2` optimization and linked against `uClibc`. The EEMBC lite benchmarks were run for the default number of iterations, and the simulator operated in user-level simulation mode so as to eliminate the effects of an underlying operating system.

The simulator itself was compiled, and the translated functions dynamically compiled, with `gcc` version 4.1.2 and `-O3` optimization. All simulations were performed on the workstation detailed in table 1 running Fedora Core 7 under conditions of minimal system load. The simulator was configured to use a simulation epoch of 1000 basic blocks and a translation threshold of 1 to ensure that every translation-unit would be translated. In addition it was setup to model target memory with a physical page size of 8KB.

6.1 Performance

Figure 4 shows the simulation performance profiles for BB, SCC, CFG and Page DBT simulation modes. Each benchmark was simulated three times, with each

Table 1. Host Configuration

Model	Dell OptiPlex
Processor	1 × Intel Core 2 Duo 6700
CPU frequency	2660 MHz
L1 caches	32KB I/D caches
L2 cache	4MB per dual-core
FSB frequency	1066 MHz
RAM	2GB, 800MHz, DDRII

successive simulation loading the translations produced by the previous simulation/s. The "outlined bars" show the percentage of total simulation time spent performing compilation. No target instructions were interpreted (zero compilation overhead) on the third simulation run, and this was the case for the majority of benchmarks on the second run.

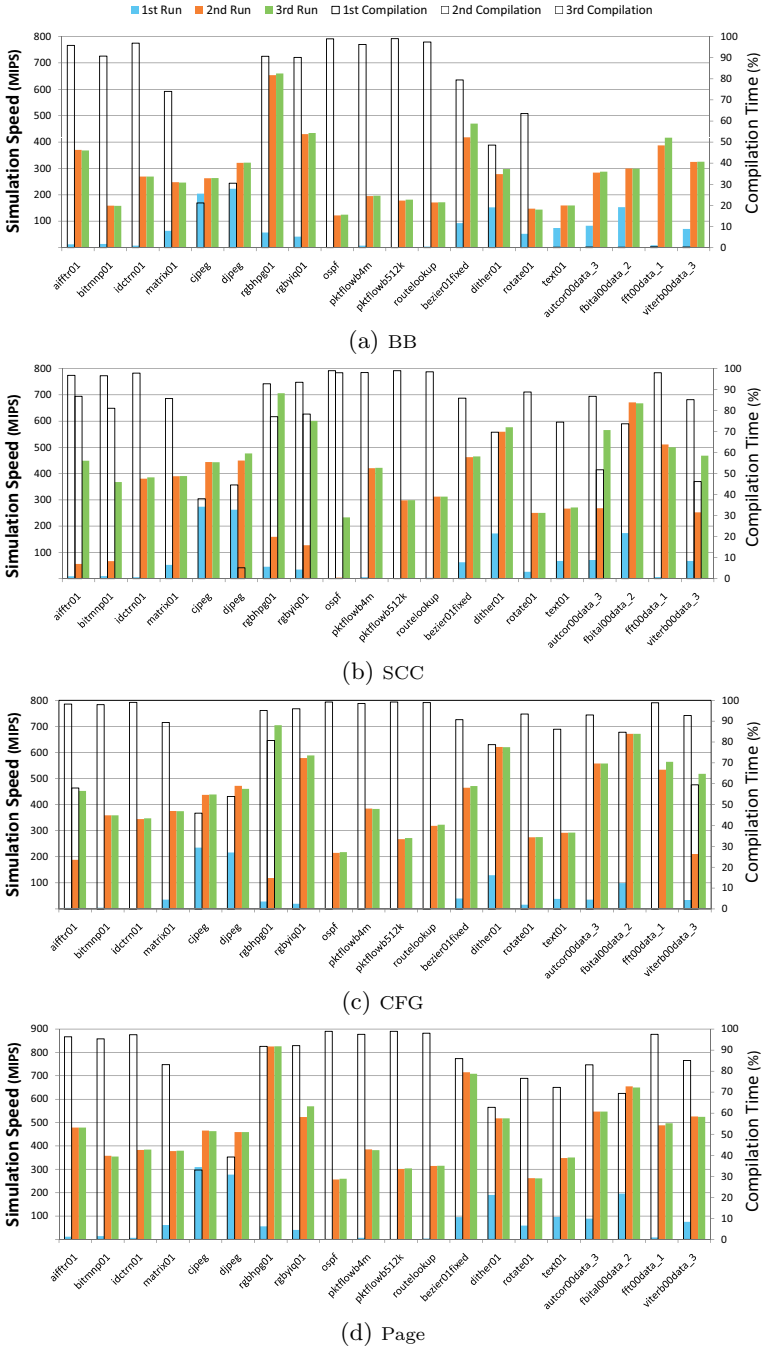
The third run represents the maximum possible simulation speed for a benchmark in a particular simulation mode. If dynamic compilation were performed in parallel to the actual simulation, then simulation speeds very close the maximum possible could be attained on the first run. As many of these benchmarks run for very short periods of time, the percentage of total simulation time spent performing compilation can be very significant (see Fig. 4). However, for longer simulations this percentage tends towards zero on the first run.

Overall, the three LTU DBT modes perform significantly better than BB DBT. The speedups for each LTU DBT mode, compared to BB DBT, are shown in Fig. 5 and summarized in table 2. LTU DBT simulation outperforms BB DBT simulation for all benchmarks with the exception of the bezierfixed benchmark, where BB DBT slightly outperforms SCC DBT. All LTU DBT simulation modes result in a mean speedup of at least 1.63 compared to BB DBT. Page DBT performs the best across all benchmarks with a mean speedup of 1.67 and standard deviation of 1.21. However, SCC DBT exhibits the highest simulation speed in 9 out of 20 of the benchmarks, compared to 7 out of 20 for Page DBT.

6.2 Translated Functions

Table 3 shows the average and largest, static and dynamic, TF block sizes broken down by benchmark. As expected, the average static TF block size for a given benchmark is greater for CFG translation than for SCC translation, and likewise greater for Page translation than for CFG translation.

Also, the average dynamic TF block size is greater for CFG based simulation than for SCC based simulation, with the exception of the pktflowb4m benchmark. In the case of pktflowb4m, CFG and Page based simulation exhibit a lower average dynamic block size than SCC simulation. This is due to the large number of TFs called in CFG and Page based simulation which simulate only a small number of basic blocks. For this benchmark, SCC simulation called 1,182 TFs which executed a single block, whereas CFG simulation called 283,038 TFs



(a) BB

(b) SCC

(c) CFG

(d) Page

Fig. 4. Different DBT Mode Simulation Profiles

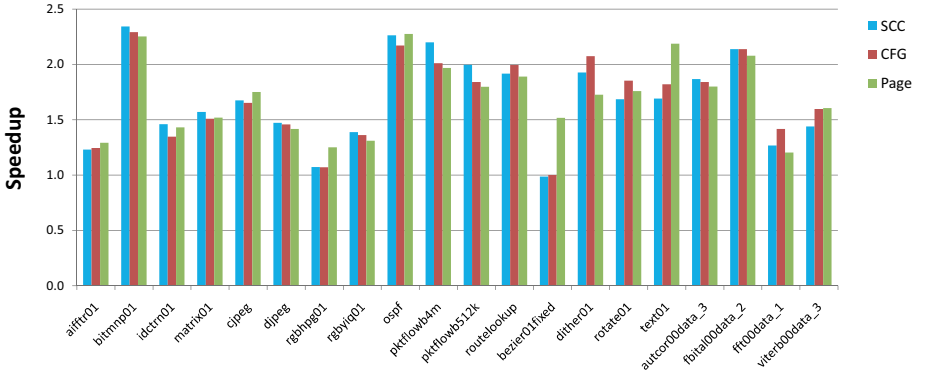


Fig. 5. Performance Comparison. This figure compares the speedups, relative to Basic Block based DBT, for each LTU simulation mode. The results shown are for the third simulation run.

Table 2. Performance Summary

	Speed (MIPS)				
	Interpretive	BB	SCC	CFG	Page
Slowest	24	124	233	217	259
Fastest	33	660	706	705	826
Median	29	278	446	445	461
Average	30	283	460	455	466
Speedup: Geo Mean	0.11	1.00	1.63	1.64	1.67
Speedup: Geo SD	1.48	1.00	1.27	1.26	1.21

The average speed is equal to the total number of instructions divided by the total simulation time for all benchmarks.

which executed a single block and Page simulation called 282,816 TFs which executed three blocks.

However, the average dynamic TF block size for Page based simulation is not greater than for CFG based simulation, the sole exception being the ospf benchmark. This stems from the fact that Page based simulation provides direct entry to any block within a TF. This means that once a block has been translated, it will always be called directly (it is never again interpreted) and therefore ceases to be profiled. This is not a problem so long as there are no program phase changes which result in new hot paths at memory addresses already translated. However, phase changes affect Page based simulation particularly severely. They can result in individual blocks, or small groups of blocks, on new hot paths being viewed as isolated code segments which will then be translated as separate TFs. This greatly reduces the average dynamic TF block size.

For a given benchmark and simulation mode there exists a TF (or multiple TFs) which represents the the simulation of the target program's main loop, or part thereof. In the case of the rgbhpg benchmark, TFs with a dynamic block size

Table 3. Translated Function Profiles

Benchmark	Avg TF Size			Avg Dynamic TF Size			Largest TF			Largest Dynamic TF		
	SCC	CFG	Page	SCC	CFG	Page	SCC	CFG	Page	SCC	CFG	Page
aifftr	3.4	7.4	16.4	4.4	6.0	5.1	87	126	134	240127	240129	5120
bitmnp	5.0	10.6	33.1	10.3	14.4	10.8	54	79	116	10111	10113	10113
idctrn	3.0	12.9	25.4	11.5	123.9	9.2	34	97	114	8193	8801	8193
matrix	3.5	6.5	28.7	5.7	11.8	9.3	45	97	174	3419	3419	3419
cjpeg	3.6	6.5	20.8	12.6	42.1	40.5	51	72	108	69129	69129	69129
djpeg	3.7	6.2	20.0	48.8	80.5	77.4	43	94	121	64327	64327	64327
rgbhpg	3.7	6.3	25.3	39.7	59.4	26.5	43	87	125	75921	75925	75925
rgbyiq	3.9	6.6	31.0	3612.9	4921.7	4272.4	43	97	133	4607999	4608002	4608002
ospf	4.4	8.7	26.0	426.3	484.6	497.3	64	105	148	189599	189602	189602
pktflowb4m	4.4	7.4	31.3	2485.1	17.8	17.8	39	90	149	1171544	1171548	1171548
pktflowb512k	4.4	7.6	31.7	325.9	397.5	17.3	39	90	149	154794	154798	154798
routelookup	4.3	7.5	27.7	667.6	712.8	37.0	41	105	105	56453	56457	56457
bezierfixed	4.0	7.2	26.2	1029.9	7163.2	1057.4	43	105	135	187799	187802	187802
dither	4.0	6.0	28.6	33.6	30332.6	33.6	44	97	130	163838	290986	163841
rotate	8.3	16.5	34.5	34.5	121.9	4.7	100	105	134	16644	16648	16648
text	4.3	10.4	28.1	4.3	10.2	9.4	43	105	136	2590	3523	2605
autcor	3.4	7.2	29.6	4350.9	5323.8	4518.1	41	86	127	15567	15573	15573
fbital	4.0	9.5	25.7	148.1	508.4	508.0	34	73	111	10752	10754	10754
fft	3.8	7.9	31.3	32.8	102.0	69.6	41	99	123	10495	10498	10498
viterb	4.3	10.7	22.1	1498.2	2617.6	1520.5	34	77	108	22475	22511	22511
Average	4.2	8.5	27.2	739.1	2652.6	637.1	48.2	94.3	128.9	354088.8	360527.3	342343.3

of 75,921 blocks were called a total of 100 times, the same as the default number of iterations for the target program. This is true for all the LTU simulation modes, suggesting that the main loop, or part thereof, for rgbhpg is contained within a strongly connected component.

6.3 Simulation Tasks

The total simulation time is divided between five main tasks. The main simulation loop calls TFs and interprets instructions; the library loading function loads previously compiled translations from the shared libraries; the page-CFG function adds interpreted blocks to the page-CFG (LTU DBT only); the profile analysis function analyses page-CFGs in the ECC (BBs in the EBC) and

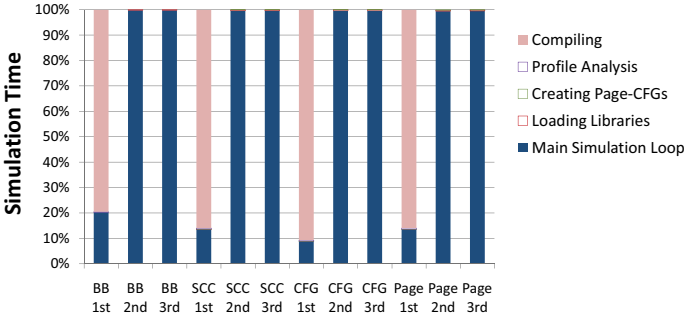


Fig. 6. Bezierfixed Simulation Tasks. This figure shows the percentage of total simulation time spent performing each of the five main tasks.

identifies hot translation-units; and the dynamic compilation function compiles hot translation-units and creates shared libraries.

Figure 6 shows how much time is spent performing each task for the bezierfixed benchmark. On the first run, 80 - 92% of the time is spent compiling the hot translation-units, with the rest of the time spent in the main simulation loop. On successive runs however, almost 100% of the time is spent in the main simulation loop. Time spent performing the other tasks is all but insignificant. This is the general pattern observed for all benchmarks. BB based simulation outperformed SCC based simulation for this benchmark because it spent an average of just 1.19 seconds in the main simulation loop, compared to 1.22 seconds for SCC.

7 Related Work

Previous work on high-speed instruction set simulation has tended to focus on compiled and hybrid mode simulators. Whilst an interpretive simulator spends most of its time repeatedly fetching and decoding target instructions, a compiled based simulator fetches and decodes each instruction once, spending most of its time performing the operations.

A statically-compiled simulator [7] which employed in-line macro expansion was shown to run up to three times faster than an interpretive simulator. Target code is statically translated to host machine code which is then executed directly within a switch statement.

Dynamic translation techniques are used to overcome the lack of flexibility inherent in statically-compiled simulators. The MIMIC simulator [6] simulates IBM System/370 instructions on the IBM RT PC. Groups of target basic blocks are translated in to host instructions, with an expansion factor of about 4 compared with natively compiled source code. On average MIMIC could simulate S/370 code at the rate of 200 instructions per second on a 2 MIPS RT PC.

Shade [4] and Embra [14] use DBT with translation caching techniques in order to increase simulation speeds. Shade is able to simulate SPARC V8, SPARC V9, and MIPS I code on a SPARC V8 platform. On average Shade simulates V8 SPEC89 integer and floating-point binaries 6.2 and 2.3 times slower respectively than they run natively. The corresponding V9 binaries are simulated 12.2 and 4 times slower respectively.

Embra, which is part of the SimOS [11] simulation environment, can simulate MIPS R3000/R4000 binary code on a Silicon Graphics IRIX machine. In its fastest configuration Embra can simulate SPEC92 benchmarks at speeds ranging from 11.1 to 20 MIPS, corresponding to slowdowns of 8.7 to 3.5. The test machine used for benchmarking was an SGI Challenge with four MIPS R4400, 150MHz processors.

More recently a number of research groups have developed retargetable instruction set simulators. The statically-compiled method exhibited in [3] applies a static scheduling technique. Whilst this increases simulation performance, it does so at the expense of flexibility. Compiled simulators were generated from model descriptions of TI's TMS320C54x and the ARM7 processors. The results showed that for the TMS320C54X processor, static scheduling gave a speedup of almost a factor of 4 when compared with dynamically scheduled simulation.

For the ARM7 processor there was an observed speedup by a factor of 7, from 5 (dynamic) to 35.5 (static) MIPS.

The Ultra-fast Instruction Set Simulator [15] improves the performance of statically-compiled simulation by using low-level binary translation techniques to take full advantage of the host architecture. Results showed that deploying static compilation with this hybrid technique lead to a simulation speedup of 3.5.

Just-In-Time Cache Compiled Simulation (JIT-CCS) [8] executes and then caches pre-compiled instruction-operation functions for each instruction fetched. The JIT-CCS simulation speed, with a reasonably large simulation cache, is within 5% of a compiled simulator, and at least 4 times faster than an equivalent interpretive simulator. The simulator was benchmarked by simulating adpcm running on ARM7 and STM ST200 functional models. Running on a 1200MHz Athlon based machine, adpcm could be simulated at up to 8 MIPS (ARM7 and ST200).

The Instruction Set Compiled Simulation (IS-CS) simulator [10] was designed to be a high performance and flexible functional simulator. To achieve this the time-consuming instruction decode process is performed during the compile stage, whilst interpretation is enabled at simulation time. Performance is further increased by a technique called instruction abstraction which produces optimized decoded instructions. A simulation speed of up to 12.2 MIPS is quoted for adpcm (ARM7 functional model) running on a 1GHz Pentium III host.

The SimICS [5] full system simulator translates the target machine-code instructions in to an intermediate format before interpretation. During simulation the intermediate format instructions are processed by the interpreter which calls the corresponding service routines. A number of SPECint95 benchmarks were simulated on a Sun Ultra Enterprise host (SunOS 5.x) resulting in a slowdown by of a factor of 26 to 75 compared with native execution.

QEMU [1,2] is a fast simulator which uses an original dynamic translator. Each target instruction is divided into a simple sequence of micro operations, the set of micro operations having been pre-compiled offline into an object file. During simulation the code generator accesses the object file and concatenates micro operations to form a host function that emulates the target instructions within a block. User-level simulation of the Linux BYTEmark benchmarks shows a slowdown of 4 for integer code and a slowdown of 10 for floating point code over native execution. System-level simulation results in a further slowdown by a factor of 2.

SimitARM [9] increases simulation speed by enabling the tasks involved in binary translation to be distributed amongst other processors. The simulator interprets target instructions, identifying frequently executed pages for DBT. When the execution count for a page exceeds a certain threshold it is compiled by GCC into a shared library which is then loaded. Simulation Results for the SPEC CINT2000 benchmarks running on a four processor host (2.8GHz P4) showed an average simulation speed of 197 MIPS for the MIPS32 ISA and 133 MIPS for the ARM v4 ISA.

8 Conclusions

This paper used the Edinburgh High Speed simulator to illustrate and compare three new DBT profiling techniques. It demonstrates that simulator performance is significantly increased by translating Large Translation Units rather than basic blocks. LTUs not only provide greater scope for optimization at compile time, they also enable more target instructions to be simulated within a TF, resulting in less time being wasted in the main simulation loop.

The EHS simulator identifies hot paths, rather than hot blocks or pages, by dynamically profiling the target program's execution path. The hot paths are then decomposed in to LTUs which are dynamically compiled and loaded. The results show all of the LTU simulation modes to be at least 1.63 times faster on average than basic block based simulation.

Page DBT simulation performs the best across all benchmarks with a mean speedup of 1.67. However, in order to attain optimal performance one would require a simulator that was capable of dynamically switching between DBT modes (BB, SCC, CFG and Page) dependent upon application behaviour and host machine architecture. It would also be desirable for a simulator to detect program phase changes so that existing TFs could be discarded in favour of translating the current hot paths.

References

1. Bartholomew, D.: QEMU: a Multihost, Multitarget Emulator. Linux Journal 2006(145), 3 (2006)
2. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: ATEC 2005: Proceedings of the USENIX Annual Technical Conference, p. 41. USENIX Association (2005)
3. Braun, G., Hoffmann, A., Nohl, A., Meyr, H.: Using Static Scheduling Techniques for the Retargeting of High Speed, Compiled Simulators for Embedded Processors from an Abstract Machine Description. In: ISSS 2001: Proceedings of the 14th International Symposium on Systems Synthesis, pp. 57–62. ACM Press, New York (2001)
4. Cmelik, B., Keppel, D.: Shade: A Fast Instruction-Set Simulator for Execution Profiling. In: SIGMETRICS 1994: Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 128–137. ACM Press, New York (1994)
5. Magnusson, P.S., Dahlgren, F., Grahn, H., Karlsson, M., Larsson, F., Lundholm, F., Moestedt, A., Nilsson, J., Stenstrm, P., Werner, B.: SimICS/sun4m: A Virtual Workstation. In: ATEC 1998: Proceedings of the Usenix Annual Technical Conference, pp. 119–130. USENIX Association (1998)
6. May, C.: Mimic: A Fast System/370 Simulator. In: SIGPLAN 1987: Papers of the Symposium on Interpreters and Interpretive Techniques, pp. 1–13. ACM Press, New York (1987)
7. Mills, C., Ahalt, S.C., Fowler, J.: Compiled Instruction Set Simulation. Journal Software, Practice and Experience 21(8), 877–889 (1991)

8. 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 2002: Proceedings of the 39th Conference on Design Automation, pp. 22–27. ACM Press, New York (2002)
9. Qin, W., D'Errico, J., Zhu, X.: A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-Compiled Instruction-Set Simulation. In: CODES+ISSS 2006: Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, pp. 193–198. ACM Press, New York (2006)
10. Reshadi, M., Mishra, P., Dutt, N.: Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. In: DAC 2003: Proceedings of the 40th Conference on Design Automation, pp. 758–763. ACM Press, New York (2003)
11. Rosenblum, M., Herrod, S.A., Witchel, E., Gupta, A.: Complete Computer System Simulation: The SimOS Approach. Journal IEEE Parallel Distrib. Technol. 3(4), 34–43 (1995)
12. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
13. Topham, N., Jones, D.: High Speed CPU Simulation using JIT Binary Translation. In: MoBS 2007: Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation (2007)
14. Witchel, E., Rosenblum, M.: Embra: Fast and Flexible Machine Simulation. In: SIGMETRICS 1996: Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 68–79. ACM Press, New York (1996)
15. Zhu, J., Gajski, D.D.: A Retargetable, Ultra-Fast Instruction Set Simulator. In: DATE 1999: Proceedings of the Conference on Design, Automation and Test in Europe, p. 62. ACM Press, New York (1999)

Integrated Modulo Scheduling for Clustered VLIW Architectures

Mattias V. Eriksson and Christoph W. Kessler

PELAB, Dept. of Computer and Information Science
Linköping university
S-58183 Linköping, Sweden
{mater,chrke}@ida.liu.se

Abstract. We solve the problem of integrating modulo scheduling with instruction selection (including cluster assignment), instruction scheduling and register allocation, with optimal spill code generation and scheduling. Our method is based on integer linear programming. We prove that our algorithm delivers optimal results in finite time for a certain class of architectures. We believe that these results are interesting both from a theoretical point of view and as a reference point when devising heuristic methods.

1 Introduction

Many computationally intensive programs spend most of their execution time in a few inner loops. This makes it important to have good methods for code generation for loops, since small improvements per loop iteration can have a large impact on overall performance.

The *back end* of a compiler transforms an intermediate representation into executable code. This transformation is usually performed in three phases: *instruction selection* selects which instructions to use, *instruction scheduling* maps each instruction to a time slot and *register allocation* selects in which registers a value is to be stored. Furthermore the back end can also contain various optimization phases, e.g. modulo scheduling for loops where the goal is to overlap iterations of the loop and thereby increase the throughput.

It is beneficial to integrate the phases of the code generation since this gives more opportunity for optimizations. However, this integration of phases comes at the cost of a greatly increased size of the solution space. In previous work [6] we gave an integer linear program formulation for integrating instruction selection, instruction scheduling and register allocation. In this paper we will show how to extend that formulation to also do modulo scheduling for loops. In contrast to earlier approaches to optimal modulo scheduling, our method aims to produce provably optimal modulo schedules with integrated cluster assignment and instruction selection.

The remainder of this paper is organized as follows: In order to give a more accessible presentation of the integer linear programming formulation for integrated modulo scheduling, we first give, in Section 2, an integer linear program for integrated code generation of basic blocks, which is adapted from earlier work [6]. In

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 65–79, 2009.

Section 3 we extend this formulation to modulo scheduling. Section 4 presents an algorithm for modulo scheduling, and proves that it is optimal for a certain class of architectures. Section 5 shows an experimental evaluation, Section 6 reviews some related work, and Section 7 concludes.

2 Integer Linear Programming Formulation

In this section we introduce the integer linear programming formulation for basic block scheduling. This model integrates instruction selection (including cluster assignment), instruction scheduling and register allocation.

2.1 Optimization Parameters and Variables

Data Flow Graph. The data flow graph of a basic block is modeled as a directed acyclic graph (DAG). The set V is the set of intermediate representation (IR) nodes, the sets $E_1, E_2 \subset V \times V$ represents edges between operations and their first and second operand respectively. $E_m \subset V \times V$ represents data dependences in memory. The integer parameters Op_i and $Outdg_i$ describe operators and out-degrees of the IR node $i \in V$, respectively.

Instruction Set. The instructions of the target machine are modeled by the set P of patterns. P consists of the set P_1 of singletons, which only cover one IR node, the set P_{2^+} of composites, which cover multiple IR nodes, and the set P_0 of patterns for non-issue instructions. The non-issue instructions are needed when there are IR nodes in V that do not have to be covered by an instruction, e.g. an IR node representing a constant value. The IR is low level enough so that all patterns model exactly one (or zero in the case of constants) instructions of the target machine.

For each pattern $p \in P_{2^+} \cup P_1$ we have a set $B_p = \{1, \ldots, n_p\}$ of generic nodes for the pattern. For composites we have $n_p > 1$ and for singletons $n_p = 1$. For composite patterns $p \in P_{2^+}$ we also have $EP_p \subset B_p \times B_p$, the set of edges between the generic pattern nodes. Each node $k \in B_p$ of the pattern $p \in P_{2^+} \cup P_1$ has an associated operator number $OP_{p,k}$ which relates to operators of IR nodes. Also, each $p \in P$ has a latency L_p, meaning that if p is scheduled at time slot t the result of p is available at time slot $t + L_p$.

Resources and Register Sets. For the integer linear programming model we assume that the functional units are fully pipelined. Hence we can model the resources of the target machine with the set $\mathcal{F}$ and the register banks by the set $\mathcal{RS}$. The binary parameter $U_{p,f,o}$ is 1 iff the instruction with pattern $p \in P$ uses the resource $f \in \mathcal{F}$ at time step o relative to the issue time. Note that this allows for multiblock [9] and irregular reservation tables [15]. $\mathcal{R}_r$ is a parameter describing the number of registers in the register bank $r \in \mathcal{RS}$. The issue width is modeled by ω, i.e. the maximum number of instructions that may be issued at any time slot.

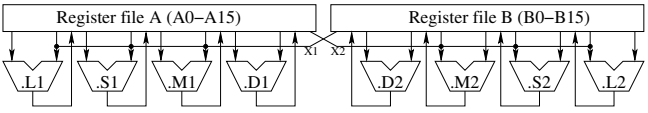


Fig. 1. The Texas Instruments TI-C62x processor has two register banks and 8 functional units [18]. The crosspaths X1 and X2 are modeled as resources, too.

For modeling transfers between register banks we do not use regular instructions (note that transfers, like spill instructions, do not cover nodes in the DAG). Instead we use the integer parameter $LX_{r,s}$ to model the latency of a transfer from $r \in \mathcal{RS}$ to $s \in \mathcal{RS}$. If no such transfer instruction exists we set $LX_{r,s} = \infty$. And for resource usage, the binary parameter $UX_{r,s,f}$ is 1 iff a transfer from $r \in \mathcal{RS}$ to $s \in \mathcal{RS}$ uses resource $f \in \mathcal{F}$. See Figure 1 for an illustration of a clustered architecture.

Lastly, we have the sets $PD_r, PS1_r, PS2_r \subset P$ which, for all $r \in \mathcal{RS}$, contain the pattern $p \in P$ iff p stores its result in r, takes its first operand from r or takes its second operand from r, respectively.

Solution Variables. The parameter $t_{\max}$ gives the last time slot on which an instruction may be scheduled. We also define the set $T = \{0, 1, 2, \ldots, t_{\max}\}$, i.e. the set of time slots on which an instruction may be scheduled. For the acyclic case $t_{\max}$ is incremented until a solution is found.

So far we have only mentioned the parameters which describe the optimization problem. Now we introduce the solution variables which define the solution space. We have the following binary solution variables:

- $c_{i,p,k,t}$, which is 1 iff IR node $i \in V$ is covered by $k \in B_p$, where $p \in P$, issued at time $t \in T$.
- $w_{i,j,p,t,k,l}$, which is 1 iff the DAG edge $(i,j) \in E_1 \cup E_2$ is covered at time $t \in T$ by the pattern edge $(k,l) \in EP_p$ where $p \in P_{2^+}$ is a composite pattern.
- $s_{p,t}$, which is 1 iff the instruction with pattern $p \in P_{2^+}$ is issued at time $t \in T$.
- $x_{i,r,s,t}$, which is 1 iff the result from IR node $i \in V$ is transfered from $r \in \mathcal{RS}$ to $s \in \mathcal{RS}$ at time $t \in T$.
- $r_{rr,i,t}$, which is 1 iff the value corresponding to the IR node $i \in V$ is available in register bank $rr \in \mathcal{RS}$ at time slot $t \in T$.

We also have the following integer solution variable:

- τ is the first clock cycle on which all latencies of executed instructions have expired.

2.2 Removing Variables by Doing Soonest-Latest Analysis

We can significantly reduce the number of variables in the model by performing soonest-latest analysis [11] on the nodes of the graph. Let $L_{\min}(i)$ be 0 if the

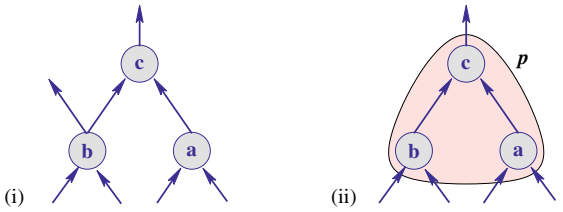


Fig. 2. (i) Pattern p can not cover the set of nodes since there is another outgoing edge from b, (ii) p covers nodes a, b, c

node $i \in V$ may be covered by a composite pattern, and the lowest latency of any instruction $p \in P_1$ that may cover the node $i \in V$ otherwise.

Let $\text{pred}(i) = \{j : (j,i) \in E\}$ and $\text{succ}(i) = \{j : (i,j) \in E\}$. We can recursively calculate the soonest and latest time slot on which node i may be scheduled:

$$soonest'(i) = \begin{cases} 0 & , \text{if } |\text{pred}(i)| = 0 \\ \max_{j \in \text{pred}(i)} \{soonest'(j) + L_{\min}(j)\} & , \text{otherwise} \end{cases} \quad (1)$$

$$latest'(i) = \begin{cases} t_{\max} & , \text{if } |\text{succ}(i)| = 0 \\ \max_{j \in \text{succ}(i)} \{latest'(j) - L_{\min}(i)\} & , \text{otherwise} \end{cases} \quad (2)$$

$$T_i = \{soonest'(i), \ldots, latest'(i)\}$$

We can also remove all the variables in c where no node in the pattern $p \in P$ has an operator number matching i. Mathematically we can say that the matrix c of variables is sparse; the constraints dealing with c must be rewritten to take this into account (basically changing $\forall i \in V,\ \forall t \in T$ to $\forall i \in V,\ \forall t \in T_i$).

2.3 Optimization Constraints

Optimization Objective. The objective of the integer linear program is to minimize the execution time:

$$\min \tau \quad (3)$$

The execution time is the latest time slot where any instruction terminates. For efficiency we only need to check for execution times for instructions covering an IR node with out-degree 0, let $V_{\text{root}} = \{i \in V : Outdg_i = 0\}$:

$$\forall i \in V_{\text{root}},\ \forall p \in P,\ \forall k \in B_p,\ \forall t \in T, \quad c_{i,p,k,t}(t + L_p) \leq \tau \quad (4)$$

Node and Edge Covering. Exactly one instruction must cover each IR node:

$$\forall i \in V, \quad \sum_{p \in P} \sum_{k \in B_p} \sum_{t \in T} c_{i,p,k,t} = 1 \quad (5)$$

Equation 6 sets $s_{p,t} = 1$ iff the composite pattern $p \in P_{2^+}$ is used at time $t \in T$. This equation also guarantees that either all or none of the generic nodes $k \in B_p$ are used at a time slot:

$$\forall p \in P_{2^+},\ \forall t \in T,\ \forall k \in B_p \quad \sum_{i \in V} c_{i,p,k,t} = s_{p,t} \tag{6}$$

An edge within a composite pattern may only be used if there is a corresponding edge (i, j) in the DAG and both i and j are covered by the pattern, see Figure 2:

$$\forall (i,j) \in E_1 \cup E_2,\ \forall p \in P_{2^+},\ \forall t \in T,\ \forall (k,l) \in EP_p,$$

$$2w_{i,j,p,t,k,l} \leq c_{i,p,k,t} + c_{j,p,l,t} \tag{7}$$

If a generic pattern node covers an IR node, the generic pattern node and the IR node must have the same operator number:

$$\forall i \in V,\ \forall p \in P,\ \forall k \in B_p,\ \forall t \in T, \quad c_{i,p,k,t}(Op_i - OP_{p,k}) = 0 \tag{8}$$

Register Values. A value may only be present in a register bank if: it was just put there by an instruction, it was available there in the previous time step, or just transfered to there from another register bank:

$$\forall rr \in \mathcal{RS},\ \forall i \in V,\ \forall t \in T,$$

$$r_{rr,i,t} \leq \sum_{\substack{p \in PD_{rr} \cap P \\ k \in B_p}} c_{i,p,k,t-L_p} + r_{rr,i,t-1} + \sum_{rs \in \mathcal{RS}} (x_{i,rs,rr,t-LX_{rs,rr}}) \tag{9}$$

The operand to an instruction must be available in the correct register bank when we use it. A limitation of this formulation is that composite patterns must have all operands and results in the same register bank:

$$\forall (i,j) \in E_1 \cup E_2,\ \forall t \in T,\ \forall rr \in \mathcal{RS},$$

$$\text{BIG} \cdot r_{rr,i,t} \geq \sum_{\substack{p \in PD_{rr} \cap P_{2^+} \\ k \in B_p}} \left(c_{j,p,k,t} - \text{BIG} \cdot \sum_{(k,l) \in EP_p} w_{i,j,p,t,k,l} \right) \tag{10}$$

where BIG is a large integer value.

Internal values in a composite pattern must not be put into a register (e.g. the multiply value in a multiply-and-accumulate instruction):

$$\forall p \in P_{2^+},\ \forall (k,l) \in EP_p,\ \forall (i,j) \in E_1 \cup E_2,$$

$$\sum_{rr \in \mathcal{RS}} \sum_{t \in T} r_{rr,i,t} \leq \text{BIG} \cdot \left(1 - \sum_{t \in T} w_{i,j,p,t,k,l} \right) \tag{11}$$

If they exist, the first operand (Equation 12) and the second operand (Equation 13) must be available when they are used:

$$\forall (i,j) \in E_1,\ \forall t \in T,\ \forall rr \in \mathcal{RS}, \quad \text{BIG} \cdot r_{rr,i,t} \geq \sum_{\substack{p \in PS1_{rr} \cap P_1 \\ k \in B_p}} c_{j,p,k,t} \tag{12}$$

$$\forall (i,j) \in E_2,\ \forall t \in T,\ \forall rr \in \mathcal{RS}, \quad \text{BIG} \cdot r_{rr,i,t} \geq \sum_{\substack{p \in PS2_{rr} \cap P_1 \\ k \in B_p}} c_{j,p,k,t} \tag{13}$$

Transfers may only occur if the source value is available:

$$\forall i \in V,\ \forall t \in T,\ \forall rr \in \mathcal{RS}, \quad r_{rr,i,t} \geq \sum_{rq \in \mathcal{RS}} x_{i,rr,rq,t} \tag{14}$$

Memory Data Dependences. Equation 15 ensures that data dependences in memory are not violated, adapted from [7]:

$$\forall (i,j) \in E_m,\ \forall t \in T \quad \sum_{p \in P} \sum_{t_j=0}^{t} c_{j,p,1,t_j} + \sum_{p \in P} \sum_{t_i=t-L_p+1}^{t_{\max}} c_{i,p,1,t_i} \leq 1 \tag{15}$$

Resources. We must not exceed the number of available registers in a register bank at any time:

$$\forall t \in T,\ \forall rr \in \mathcal{RS}, \quad \sum_{i \in V} r_{rr,i,t} \leq R_{rr} \tag{16}$$

Condition 17 ensures that no resource is used more than once at each time slot:

$$\forall t \in T,\ \forall f \in \mathcal{F},$$

$$\sum_{\substack{p \in P_{2^+} \\ o \in \mathbb{N}}} U_{p,f,o} s_{p,t-o} + \sum_{\substack{p \in P_1 \\ i \in V \\ k \in B_p}} U_{p,f,o} c_{i,p,k,t-o} + \sum_{\substack{i \in V \\ (rr,rq) \in (\mathcal{RS} \times \mathcal{RS})}} UX_{rr,rq,f} x_{i,rr,rq,t} \leq 1 \tag{17}$$

And, lastly, Condition 18 guarantees that we never exceed the issue width:

$$\forall t \in T, \quad \sum_{p \in P_{2^+}} s_{p,t} + \sum_{\substack{p \in P_1 \\ i \in V \\ k \in B_p}} c_{i,p,k,t} + \sum_{\substack{i \in V \\ (rr,rq) \in (\mathcal{RS} \times \mathcal{RS})}} x_{i,rr,rq,t} \leq \omega \tag{18}$$

3 Extending the Model to Modulo Scheduling

Software pipelining [3] is an optimization for loops where the following iterations of the loop are initiated before the current iteration is finished. One well known kind of software pipelining is *modulo scheduling* [16] where new iterations of the loop are issued at a fixed rate determined by the *initiation interval*. For every

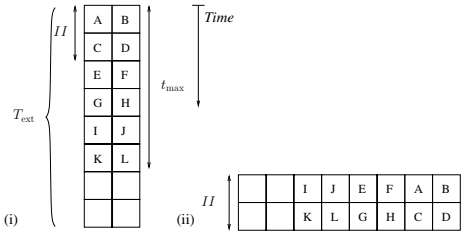


Fig. 3. An example showing how an acyclic schedule (i) can be rearranged into a modulo schedule (ii), A-L are target instructions

loop the initiation interval has a lower bound $MinII = \max(ResMII, RecMII)$, where *ResMII* is the bound determined by the available resources of the processor, and *RecMII* is the bound determined by the critical dependence cycle in the dependence graph describing the loop body. Methods for calculating *RecMII* and *ResMII* are well documented in e.g. [10].

With some work the model in Section 2 can be extended to also integrate modulo scheduling. We note that a kernel can be formed from the schedule of a basic block by scheduling each operation modulo the initiation interval, see Figure 3. The modulo schedules that we create have a corresponding acyclic schedule, and by the length of a modulo schedule we mean $t_{\max}$ of the acyclic schedule. We also note that creating a valid modulo schedule only adds constraints compared to the basic block case.

First we need to model loop carried dependences by adding a distance: E_1, E_2, $E_m \subset V \times V \times \mathbb{N}$. The element (i, j, d) represents a dependence from i to j which spans over d loop iterations. Obviously the graph is no longer a DAG since it may contain cycles. The only thing we need to do to include loop distances in the model is to change $r_{rr,i,t}$ to: $r_{rr,i,t+d \cdot II}$ in Equations 10, 12 and 13, and modify Equation 15 to

$$\forall (i,j,d) \in E_m,\ \forall t \in T_{\text{ext}} \quad \sum_{p \in P} \sum_{t_j=0}^{t-II \cdot d} c_{j,p,1,t_j} + \sum_{p \in P} \sum_{t_i=t-L_p+1}^{t_{\max}+II \cdot d_{\max}} c_{i,p,1,t_i} \leq 1 \qquad (19)$$

For this to work the initiation interval II must be a parameter to the solver. To find the best initiation interval we must run the solver several times with different values of the parameter. A problem with this approach is that it is difficult to know when an optimal II is reached if the optimal II is not *RecMII* or *ResMII*; we will get back to this problem in Section 4.

The slots on which instructions may be scheduled are defined by $t_{\max}$, and we do not need to change this for the modulo scheduling extension to work. But when we model dependences spanning over loop iterations we need to add extra time slots to model that variables may be alive after the last instruction of an iteration is scheduled. This extended set of time slots is modeled by the set $T_{\text{ext}} = \{0, \ldots, t_{\max} + II \cdot d_{\max}\}$ where $d_{\max}$ is the largest distance in any of E_1 and E_2. We extend the variables in $x_{i,r,s,t}$ and $r_{rr,i,t}$ so that they have $t \in T_{\text{ext}}$

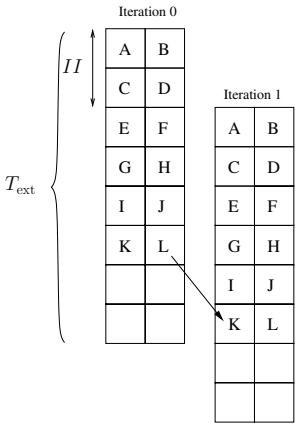


Fig. 4. An example showing why T_{ext} has enough time slots to model the extended live ranges. Here $d_{\max} = 1$ and $II = 2$ so any live value from from Iteration 0 can not live after time slot $t_{\max} + II \cdot d_{\max}$ in the acyclic schedule.

instead of $t \in T$, this is enough since a value created by an instruction scheduled at any $t \leq t_{\max}$ will be read, at latest, by an instruction $d_{\max}$ iterations later, see Figure 4 for an illustration.

3.1 Resource Constraints

The inequalities in the previous section now only need a few further modifications to also do modulo scheduling. The resource constraints of the kind $\forall t \in T,\ \text{expr} \leq \text{bound}$ become $\forall t_o \in \{0, 1, \ldots, II - 1\},\ \sum_{\substack{t \in T_{\text{ext}} \\ t \equiv t_o (\text{mod } II)}} \text{expr} \leq \text{bound}$.

For instance, Inequality 16 becomes

$$\forall t_o \in \{0, 1, \ldots, II - 1\},\ \forall rr \in \mathcal{RS},\quad \sum_{i \in V} \sum_{\substack{t \in T_{\text{ext}} \\ t \equiv t_o (\text{mod } II)}} r_{rr,i,t} \leq R_{rr} \tag{20}$$

Inequalities 17 and 18 are modified in the same way.

Inequality 20 guarantees that the number of live values in each register bank does not exceed the number of available registers. However if there are overlapping live ranges, i.e. when a value i is saved at t_d and used at $t_u > t_d + II \cdot k_i$ for some positive integer $k_i > 1$ the values in consecutive iterations can not use the same register for this value. We may solve this by doing *variable modulo expansion* [10].

3.2 Removing More Variables

As we saw in Section 2.2 it is possible to improve the solution time for the integer linear programming model by removing variables whose values can be inferred.

Input: A graph of IR nodes $G = (V, E)$, the lowest possible initiation interval *MinII*, and the architecture parameters
Output: Modulo schedule.
$MaxII = t_{\text{upper}} = \infty$;
$t_{\max} = MinII$;
while $t_{\max} \leq t_{\text{upper}}$ **do**
  Compute *soonest′* and *latest′* with the current $t_{\max}$;
  $II = MinII$;
  while $II < \min(t_{\max}, MaxII)$ **do**
    solve integer linear program instance;
    if solution found **then**
      if $II == MinII$ **then**
        return solution; *//This solution is optimal*
      fi
      $MaxII = II - 1$; *//Only search for better solutions.*
    fi
    $II = II + 1$
  od
  $t_{\max} = t_{\max} + 1$
od

Fig. 5. Pseudocode for integrated modulo scheduling

Now we can take loop-carried dependences into account and find improved bounds:

$$soonest(i) = \max\{soonest'(i), \max_{\substack{(j,i,d)\in E \\ d\neq 0}} (soonest'(j) + L_{\min}(j) - II \cdot d)\} \quad (21)$$

$$latest(i) = \max\{latest'(i), \max_{\substack{(i,j,d)\in E \\ d\neq 0}} \big(latest'(j) - L_{\min}(i) + II \cdot d\big)\} \quad (22)$$

With these new derived parameters we create

$$T_i = \{soonest(i), \ldots, latest(i)\}$$

that we can use instead of the set T for the variable $c_{i,p,k,t}$.

Equations 21 and 22 differ from Equations 1 and 2 in two ways: they are not recursive and they need information about the initiation interval. Hence, *soonest′* and *latest′* can be calculated when $t_{\max}$ is known, before the integer linear program is run, while *soonest* and *latest* can be derived parameters.

4 The Algorithm

Figure 5 shows the algorithm for finding a modulo schedule. Note that if there is no solution with initiation interval *MinII* this algorithm never terminates (we do not consider cases where $II > t_{\max}$). In the next section we will show how to make the algorithm terminate with optimal result.

A valid alternative to this algorithm would be to set $t_{\max}$ to a fixed sufficiently large value and then look for the minimal II. A problem with this approach is that the solution time of the integer linear program increases superlinearly with $t_{\max}$. Therefore we find that beginning with a low value of $t_{\max}$ and increasing it works best.

4.1 Theoretical Properties

In this section we will have a look at the theoretical properties of Algorithm 5 and show how the algorithm can be modified so that it finds optimal modulo schedules in finite time for a certain class of architectures.

Definition 1. *We say that a schedule s is* dawdling *if there is a time slot $t \in T$ such that (a) no instruction in s is issued at time t, and (b) no instruction in s is running at time t, i.e. has been issued earlier than t, occupies some resource at time t, and delivers its result at the end of t or later [9].*

Definition 2. *The* slack window *of an instruction i in a schedule s is a sequence of time slots on which i may be scheduled without interfering with another instruction in s. And we say that a schedule is* n-dawdling *if each instruction has a slack window of at most n positions.*

Definition 3. *We say that an architecture is* transfer free *if all instructions except NOP must cover a node in the IR graph. I.e., no extra instructions such as transfers between clusters may be issued unless they cover IR nodes. We also require that the register file sizes of the architecture are unbounded.*

Lemma 1. *For a transfer free architecture every non-dawdling schedule for the data flow graph (V, E) has length*

$$t_{\max} \leq \sum_{i \in V} \hat{L}(i)$$

where $\hat{L}(i)$ is the maximal latency of any instruction covering IR node i (composite patterns need to replicate $\hat{L}(i)$ over all covered nodes).

Proof. Since the architecture is transfer free only instructions covering IR nodes exist in the schedule, and each of these instructions are active at most $\hat{L}(i)$ time units. Furthermore we never need to insert dawdling NOPs to satisfy dependences of the kind $(i, j, d) \in E$; consider the two cases:

(a) $t_i \leq t_j$: Let $L(i)$ be the latency of the instruction covering i. If there is a time slot t between the point where i is finished and j begins which is not used for another instruction then t is a dawdling time slot and may be removed without violating the lower bound of j: $t_j \geq t_i + L(i) - d \cdot II$, since $d \cdot II \geq 0$.
(b) $t_i > t_j$: Let $L(i)$ be the latency of the instruction covering i. If there is a time slot t between the point where j ends and the point where i begins which is not used for another instruction this may be removed without violating the upper bound of i: $t_i \leq t_j + d \cdot II - L(i)$. ($t_i$ is decreased when removing the

dawdling time slot.) This is where we need the assumption of unlimited register files, since decreasing t_i increases the live range of i, possibly increasing the register need of the modulo schedule. □

Corollary 1. *An n-dawdling schedule for the data flow graph (V, E) has length*

$$t_{\max} \leq \sum_{i \in V} (\hat{L}(i) + n - 1).$$

Lemma 2. *If a modulo schedule s with initiation interval II has an instruction i with a slack window of size at least $2II$ time units, then s can be shortened by II time units and still be a modulo schedule with initiation interval II.*

Proof. If i is scheduled in the first half of its slack window the last II time slots in the window may be removed and all instructions will keep their position in the modulo reservation table. Likewise, if i is scheduled in the last half of the slack window the first II time slots may be removed. □

Theorem 1. *For a transfer free architecture, if there does not exist a modulo schedule with initiation interval $\tilde{II}$ and $t_{\max} \leq \sum_{i \in V} (\hat{L}(i) + 2\tilde{II} - 1)$ there exists no modulo schedule with initiation interval $\tilde{II}$.*

Proof. Assume that there exists a modulo schedule s with initiation interval $\tilde{II}$ and $t_{\max} > \sum_{i \in V} (\hat{L}(i) + 2\tilde{II} - 1)$. Also assume that there exists no modulo schedule with the same initiation interval and $t_{\max} \leq \sum_{i \in V} (\hat{L}(i) + 2\tilde{II} - 1)$. Then, by Lemma 1, there exists an instruction i in s with a slack window larger than $2\tilde{II} - 1$ and hence, by Lemma 2, s may be shortened by $\tilde{II}$ time units and still be a modulo schedule with the same initiation interval. If the shortened schedule still has $t_{\max} > \sum_{i \in V} (\hat{L}(i) + 2\tilde{II} - 1)$ it may be shortened again, and again, until the resulting schedule has $t_{\max} \leq \sum_{i \in V} (\hat{L}(i) + 2\tilde{II} - 1)$. □

Corollary 2. *We can guarantee optimality in the algorithm in Section 4 for transfer free architectures if every time we find an improved II, set $t_{\text{upper}} = \sum_{i \in V} (\hat{L}(i) + 2(II - 1) - 1)$.*

5 Experiments

The experiments were run on a computer with an Athlon X2 6000+ processor with 4 GB RAM. The version of CPLEX is 10.2.

5.1 A Contrived Example

First let us consider an example that demonstrates how Corollary 2 can be used. Figure 6 shows a graph of an example program with 4 multiplications. Consider the case where we have a non-clustered architecture with one functional unit which can perform pipelined multiplications with latency 2. Clearly, for this example we have $RecMII = 6$ and $ResMII = 4$, but an initiation interval of 6

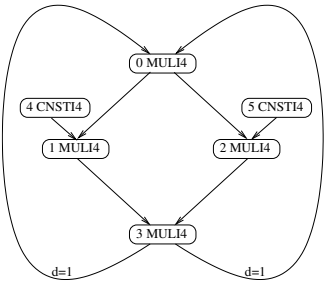


Fig. 6. A loop body with 4 multiplications. The edges between Node 3 and Node 0 are loop carried dependences with distance 1.

is impossible since IR-nodes 1 and 2 may not be issued at the same clock cycle. When we run the algorithm we quickly find a modulo schedule with initiation interval 7, but since this is larger than *MinII* the algorithm can not determine if it is the optimal solution. Now we can use Corollary 2 to find that an upper bound of 18 can be set on $t_{\max}$. If no improved modulo schedule is found where $t_{\max} = 18$ then the modulo schedule with initiation interval 7 is optimal. This example is solved to optimality in 18 seconds by our algorithm.

5.2 DSPSTONE Kernels

Table 1 shows the results of our experiments with the algorithm from Section 4. We used 5 target architectures, all variations of the Texas Instruments TI-C62x DSP processor [18]:

- single: single cluster, no MAC, no transfers and no spill,
- trfree: 2 clusters, no MAC, no transfers and no spill,
- mac: 2 clusters, with MAC, no transfers and no spill,
- mac_tr: 2 clusters, with MAC and transfers, no spill,
- mac_tr_spill: 2 clusters, with MAC, transfers and spill.

The kernels are taken from the DSPSTONE benchmark suite [20] and the dependence graphs were generated by hand. The columns marked *II* shows the found initiation interval and the columns marked $t_{\max}$ shows the schedule length. The IR does not contain branch instructions, and the load instructions are side effect free (i.e. no auto increment of pointers).

The time limit for each individual integer linear program instance was set to 3600 seconds and the time limit for the entire algorithm was set to 7200 seconds. If the algorithm timed out before an optimal solution was found the largest considered schedule length is displayed in the column marked To. We see in the results that the algorithm finds optimal results for the dotp, fir and biquad_N kernels within minutes for all architectures. For the n_complex_updates kernel an optimal solution for the single cluster architecture is found and for the other architectures the algorithm times out before we can determine if the found modulo schedule is optimal. Also for the iir kernel the algorithm times out long before we can rule out the existence of better modulo schedules.

Table 1. Experimental results with 5 DSPSTONE kernels on 5 different architectures

Architecture	*MinII*	*II*	$t_{\max}$	time(s)	To
single	5	5	17	36	-
trfree	3	3	16	41	-
mac	3	3	15	51	-
mac_tr	3	3	15	51	-
mac_tr_spill	3	3	15	55	-

(a) dotp, $|V| = 14$

Architecture	*MinII*	*II*	$t_{\max}$	time(s)	To
single	6	6	19	45	-
trfree	3	3	16	33	-
mac	3	3	16	63	-
mac_tr	3	3	16	65	-
mac_tr_spill	3	3	16	69	-

(b) fir, $|V| = 20$

Architecture	*MinII*	*II*	$t_{\max}$	time(s)	To
single	8	8	12	15	-
trfree	4	6	10	23	13
mac	4	6	10	65	12
mac_tr	4	6	10	1434	11
mac_tr_spill	4	6	10	2128	11

(c) n_complex_updates, $|V| = 27$

Architecture	*MinII*	*II*	$t_{\max}$	time(s)	To
single	9	9	14	20	-
trfree	5	5	13	33	-
mac	5	5	13	63	-
mac_tr	5	5	13	92	-
mac_tr_spill	5	5	13	191	-

(d) biquad_N, $|V| = 30$

Architecture	*MinII*	*II*	$t_{\max}$	time(s)	To
single	18	21	35	179	39
trfree	18	19	31	92	72
mac	17	19	31	522	35
mac_tr	17	20	29	4080	30
mac_tr_spill	17	20	29	4196	30

(e) iir, $|V| = 38$

We can conclude from these experiments that while the algorithm in Section 4 theoretically produces optimal results for transfer free architectures with the t_{upper} bound of Corollary 2, it is not realistic to use for even medium sized kernels because of the time required to solve big integer linear programming instances. However, in all cases, the algorithm finds a schedule with initiation interval within 3 of the optimum.

6 Related Work

Ning and Gao [13] formulated an optimal method for modulo scheduling with focus on register allocation. Altman et al. [2] presented an optimal method for simultaneous modulo scheduling and mapping of instructions to functional units. Cortadella et al. have presented an integer linear programming method for finding optimal modulo schedules [5]. These formulations works only for non-clustered architectures and do not include instruction selection. Nagarakatte and Govindarajan [12] formulated an optimal method for integrating register allocation and spill code generation.

Heuristic methods for modulo scheduling on clustered architectures have been presented by Stotzer and Leiss [17], and by Codina et al. [4]. Other notable heuristics, which are not specifically targeted for clustered architectures, are due to: Llosa et al. [11], Huff [8], and Pister and Kästner [14].

The work presented in this paper is different from the ones mentioned above in that it aims to produce provably optimal modulo schedules, also when the optimal initiation interval is larger than *MinII*, and in that it integrates also cluster assignment and instruction selection in the formulation.

Creating an integer linear programming formulation for clustered architectures is more difficult than for the non-clustered case since the common method of modeling live ranges simply as the time between definition and use can not be applied. Our formulation does it instead by a novel method where values are explicitly assigned to register banks for each time slot. This greatly increases the solution space, but we believe that this extra complexity is unavoidable and inherent to the problem of integrating cluster assignment and instruction selection with the other phases.

Touati [19] presented several theoretical results regarding the register need in modulo schedules. One of the results shows that, in the absence of resource conflicts, there exists a finite schedule duration ($t_{\max}$ in our terminology) that can be used to compute the minimal periodic register sufficiency of a loop for all its valid modulo schedules. Theorem 1 in this paper is related to this result of Touati. We assume unbounded register files and identify an upper bound on schedule duration, in the presence of resource conflicts.

7 Conclusions and Future Work

We have presented an optimal algorithm for modulo scheduling where cluster assignment, instruction selection, instruction scheduling, register allocation and spill code generation are integrated. For cases where the optimal initiation interval is larger than *MinII* we have presented a way to prove that the found *II* is optimal for transfer free architectures.

In 16 out of the 25 tests our algorithm found the optimal initiation interval. In the other 9 cases we found initiation intervals but the algorithm timed out before ruling out the existence of better modulo schedules.

A topic for future work is to compare the performance of our fully integrated method to methods where instruction selection and cluster assignment are not integrated with the other phases. We also want to find ways to relax the constraints on the machine model in the theoretical section.

Acknowledgements. This work was supported by VR and the CUGS graduate school. We thank the anonymous reviewers for their constructive comments.

References

1. Allan, V.H., Jones, R.B., Lee, R.M., Allan, S.J.: Software pipelining. ACM Comput. Surv. 27(3), 367–432 (1995)
2. Altman, E.R., Govindarajan, R., Gao, G.R.: Scheduling and mapping: Software pipelining in the presence of structural hazards. In: Proc. SIGPLAN 1995 Conf. on Programming Language Design and Implementation, pp. 139–150 (1995)

3. Charlesworth, A.E.: An approach to scientific array processing: The architectural design of the AP-120b/FPS-164 family. Computer 14(9), 18–27 (1981)
4. Codina, J.M., Sánchez, J., González, A.: A unified modulo scheduling and register allocation technique for clustered processors. In: PACT 2001: Proc. 2001 Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 175–184. IEEE Computer Society, Los Alamitos (2001)
5. Cortadella, J., Badia, R.M., Sánchez, F.: A mathematical formulation of the loop pipelining problem. In: XI Conference on Design of Integrated Circuits and Systems, Barcelona, pp. 355–360 (1996)
6. Eriksson, M.V., Skoog, O., Kessler, C.W.: Optimal vs. heuristic integrated code generation for clustered VLIW architectures. In: SCOPES, pp. 11–20 (2008)
7. Gebotys, C.H., Elmasry, M.I.: Global optimization approach for architectural synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 12(9), 1266–1278 (1993)
8. Huff, R.A.: Lifetime-sensitive modulo scheduling. In: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 258–267 (1993)
9. Kessler, C., Bednarski, A., Eriksson, M.: Classification and generation of schedules for VLIW processors. Concurrency and Computation: Practice and Experience 19(18), 2369–2389 (2007)
10. Lam, M.: Software pipelining: an effective scheduling technique for VLIW machines. SIGPLAN Not. 23(7), 318–328 (1988)
11. Llosa, J., Gonzalez, A., Ayguade, E., Valero, M.: Swing modulo scheduling: a lifetime-sensitive approach. In: Proceedings of the Conference on Parallel Architectures and Compilation Techniques, pp. 80–86 (October 1996)
12. Nagarakatte, S.G., Govindarajan, R.: Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 integer linear programming formulation. In: Krishnamurthi, S., Odersky, M. (eds.) CC 2007. LNCS, vol. 4420, pp. 126–140. Springer, Heidelberg (2007)
13. Ning, Q., Gao, G.R.: A novel framework of register allocation for software pipelining. In: POPL 1993: Proceedings of the 20th ACM symp. on Principles of programming languages, pp. 29–42. ACM, New York (1993)
14. Pister, M., Kästner, D.: Generic software pipelining at the assembly level. In: SCOPES 2005: Proc. workshop on Software and compilers for embedded systems, pp. 50–61. ACM, New York (2005)
15. Rau, B.R.: Iterative modulo scheduling: An algorithm for software pipelining loops. In: MICRO-27, pp. 63–74 (November 1994)
16. Rau, B.R., Glaeser, C.D.: Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. SIGMICRO Newsl. 12(4), 183–198 (1981)
17. Stotzer, E., Leiss, E.: Modulo scheduling for the TMS320C6x VLIW DSP architecture. SIGPLAN Not. 34(7), 28–34 (1999)
18. Texas Instruments Incorporated. TMS320C6000 CPU and Instruction Set Reference Guide (2000)
19. Touati, S.-A.-A.: On periodic register need in software pipelining. IEEE Trans. Comput. 56(11), 1493–1504 (2007)
20. živojnović, V., Velarde, J.M., Schläger, C., Meyr, H.: DSPSTONE: A DSP-oriented benchmarking methodology. In: Proc. Int. Conf. on Signal Processing and Technology (ICSPAT 1994) (1994)
21. Wilson, T.C., Grewal, G.W., Banerji, D.K.: An ILP Solution for Simultaneous Scheduling, Allocation, and Binding in Multiple Block Synthesis. In: Proc. Int. Conf. on Computer Design (ICCD), pp. 581–586 (1994)

Software Pipelining in Nested Loops with Prolog-Epilog Merging

Mohammed Fellahi and Albert Cohen

Alchemy Group, INRIA Saclay, France, and HiPEAC Network
first.last@inria.fr

Abstract. Software pipelining (or modulo scheduling) is a powerful back-end optimization to exploit instruction and vector parallelism. Software pipelining is particularly popular for embedded devices as it improves the computation throughput without increasing the size of the inner loop kernel (unlike loop unrolling), a desirable property to minimize the amount of code in local memories or caches. Unfortunately, common media and signal processing codes exhibit *series* of *low-trip-count* inner loops. In this situation, software pipelining is often not an option: it incurs severe fill/drain time overheads and code size expansion due to nested prologs and epilogs.

We propose a method to pipeline *series of inner* loops without increasing the size of the loop nest, apart from an *outermost* prolog and epilog. Our method achieves significant code size savings and allows pipelining of low-trip-count loops. These benefits come at the cost of additional scheduling constraints, leading to a linear optimization problem to trade memory usage for pipelining opportunities.

1 Introduction

Software pipelining optimizes the exploitation of instruction-level parallelism (ILP) inside inner loops, with a reduced code size overhead compared to aggressive loop unrolling [24]. Even better, pipelining alone (without additional unrolling for register allocation) does not increase loop *kernel* size: typically, due to tighter packing of instructions, kernel size typically decreases on VLIW processors.

Unfortunately, most applications contain multiple hot inner loops occurring inside repetitive control structures; this is typically the case of streaming applications in media (or signal) processing, nesting sequences of filters (or transforms) inside long-lived outer loops (e.g., the so-called *time* loop). As streaming applications are generally bandwidth bound, targeting many-core processors with a high performance to off-chip-communication ratio, fitting more inner loops on a single cache generally means higher performance.[1] As a result, current compilers have to trade ILP in inner loops for scratch-pad resources, leading to suboptimal performance.

[1] On multi-core architectures like the IBM Cell, although all efforts where made to maximize inter-core/scratch-pad connectivity, intra-core register bandwidth is still an order of magnitude higher.

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 80–94, 2009.

2 Problem Statement

For the sake of simplicity, we will first consider two levels of nested loops, with a single outer loop, called the *global loop*, enclosing one or more inner loops, called *phases*.[2] We show that the conflict between pipelining and code size can often be a side-effect of separating the optimization of individual inner loops. *We show how to pipeline all phases without any overhead on the size of the global outer loop.*

Technically, we propose to modulo-schedule [24] phases while merging the prolog of each outer iteration of a phase with the epilog of its previous outer iteration. Such prolog-epilog merging is enabled by an outer loop retiming (or shifting) [15,5] step, at the cost of a few additional constraints on modulo scheduling. It is then possible to reintegrate the merged code block within the pipelined kernel, restoring the loop to its original number of iterations. This operation is not always possible, and depends on the outer loop's dependence cycles. Indeed, after software-pipelining, prolog-epilog merging may affect phases that are in dependence with statements shifted by the software pipeline (along an inner loop). This makes our problem more difficult than in the perfectly nested case [26].

We consider low-level code after instruction selection but before register allocation; we thus ignore scalar dependences except def-use relations, and break inductive def-use paths whenever a simple closed form exists. We assume all dependences are uniform, modeled as constant distance vectors whose dimension is the common nesting depth between the source and sink of the dependence [1]. We capture all dependences in a directed graph G labeled with (multidimensional) dependence vectors.

2.1 Running Example

Our running example is given in Fig. 2. Statements and phases are labeled. Both intra-phase and inter-phase dependence vectors are shown.

The classical approach to the optimization of such an example is (1) to look for high-level loop fusions that may improve the locality in inner loops, often resulting in array contraction and scalar promotion opportunities [1,29], and (2) to pipeline the phases (inner loops) whose trip count is high enough. We assume the first loop fusion step has been applied, and that further fusion is hampered by complex dependence patterns or mismatching loop trip counts. The result of the second step is sketched using statement labels in Fig. 3; notice the *modified termination condition* in pipelined phases. As expected, this improves ILP, at the expense of code size and some loss of ILP in each phase's prolog/epilog.

The alternative is to shift the prolog of each pipelined phase, advancing it by one iteration of the global loop, then to merge it with the corresponding epilog of the previous iteration of the global loop. This is not always possible, and we will show in the following sections how to formalize the selection of phases subject to pipelining as a linear optimization problem.

[2] An analogy with a stream-oriented coarse grain scheduling algorithm [13].

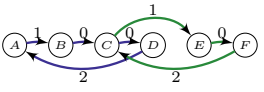


Fig. 1. Phase dependence graph

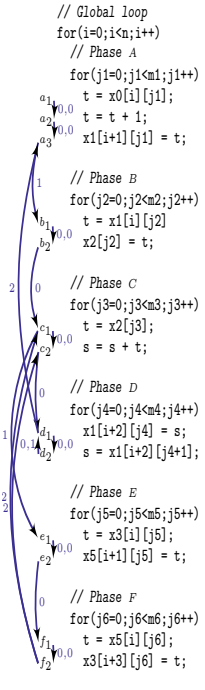


Fig. 2. Running example

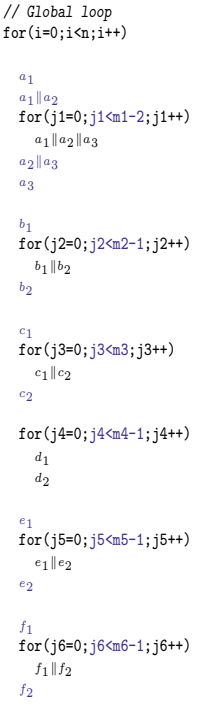


Fig. 3. Pipelining all phases

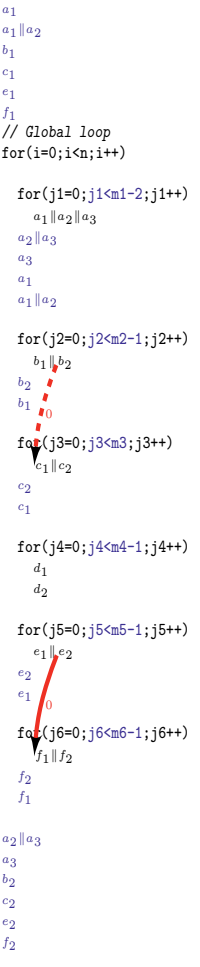


Fig. 4. Incorrect anticipation

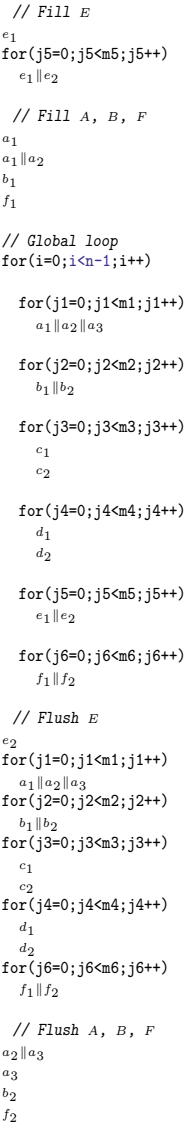


Fig. 5. Prolog-epilog merging

Back to our running example, a possible solution is to pipeline and apply prolog-epilog merging to phases A, B, E and F. The code after advancing the prologs of pipelined phases is outlined in Fig. 4; notice the outermost prolog—resulting from advancing the first global iteration of the phase prologs — and epilog — the last global iteration of phase epilogs. Yet this code is incorrect, for two reasons.

- The inter-phase dependence from statement e_2 to statement f_1 is violated, since f_1 in the prolog of phase F has been anticipated before one full iteration of phase E; some instances of this violation are depicted by a bold arc on Fig. 4. To fix this violation, one may shift the whole phase E, advancing it by one iteration of the global loop. This is possible since the only inter-phase dependence targeting phase E (statement e_1) has a non-null distance.
- A similar problem exists with the inter-phase dependence from statement b_2 to statement c_1; some instances of this violation are depicted by a bold dashed arc on Fig. 4. Yet we will see that this violation cannot be fixed by shifting, due to the accumulation of shifting constraints on the cycle of inter-phase dependences involving A, B, C and D. We choose not to pipeline C in the following; we will later demonstrate the optimality of this choice after formalizing the global optimization problem.

The final code after pipelining all phases but C,[3] prolog-epilog merging, shifting E, and collapsing the merged prologs and epilogs into the kernels is outlined in Fig. 5; notice the *modified trip count* of the global loop, and the *restored trip count* of the pipelined phases (due to prolog-epilog merging).

The body of the global loop recovered its original size, and prolog/epilog overhead has disappeared. This major improvement was done at the minor expense of the loss of ILP on phase D, and some extra code outside the global loop, due to the global shifting of phase E.[4]

2.2 Inter-phase Dependences

In the following, shifting is understood as *advancing* the execution of a statement by one or more iterations. For example, shifting b_1 implies that the first iteration of b_1 (or more) will end up in a prolog of phase B; this prolog will have to be merged with the epilog of this phase for the previous iteration of the outer loop. Since the dependence from a_3 to b_1 is carried by the outer loop, its associated distance (0) does not tell anything about the precise iterations of b_1 within phase B that are in dependence. Shifting b_1 along the inner loop — by any positive amount — is thus equivalent to shifting the whole phase B by 1 iteration of the outer loop. This observation is key to converting our prolog-epilog merging problem into a classical retiming one.

3 Characterization of Pipelinable Phases

From the global dependence graph G with multidimensional dependence vectors, the *phase dependence graph* G_p is defined as follows: nodes of G_p are the phases; an arc links a phase A to a phase B if and only if there is a path in G from a statement a of A to statement b of B; to avoid spurious transitively covered arcs, we also require this path to contain a *single inter-phase arc*; the distance

[3] Attempting to pipeline D does not bring any ILP.

[4] The first iteration of the global loop executes E only, while the last iteration executes every phase but E.

associated with an arc of G_p is the sum of the distances on the dimension of the global loop, along the corresponding path from a to b in G.

Arcs in G_p will be called *phase dependences*. Each one correspond to one inter-phase dependence and zero or more transitively-covered intra-phase dependences; the distance of a phase dependence accounts for distances along the outer dimension of intra-phase dependences.

Fig. 1 shows the phase dependence graph for the running example.

3.1 Causality Condition

Every time a phase is software pipelined, we just showed that merging its prolog and epilog is equivalent — when considering G_p — to shifting the whole phase by 1. To guarantee that all phases can be pipelined and their prolog and epilog merged, it is thus sufficient that every forward arc (p, p') in G_p has distance $d_{p,p'} > 0$, and any backward arc has distance $d_{p,p'} > 1$.

This is of course too restrictive, and in general we are back to a traditional retiming problem [15].

Let us define

$$k_C \stackrel{\text{def}}{=} \sum_{(p,p')\in C} d_{p,p'} - nb\_backward\_edges(C). \tag{1}$$

Theorem 1. *For a given cycle, up to k_C phases can be pipelined with prolog-epilog merging.*

Let us prove this result. Let $a(i, p, j)$ denote an instance of instruction a, given an iteration i of the global loop, a phase p and an iteration j of p. Let $t_{a(i,p,j)}$ denote the execution time of $a(i, p, j)$ and $a(i, p)$ denote the set of instances of a at global loop iteration i. Assuming $b(i', p')$ depends on $a(i, p)$ with dependence distance d, we have

$$\forall j, j', t_{a(i,p,j)} < t_{b(i',p',j')} \text{ and } i \leq i'.$$

Indeed, a phase dependence in G_p between p and p' corresponds to dependences between two sets of statement instances $a(i, p)$ and $b(i', p')$.

Software pipelining p' may require shifting occurrences of instruction a. We call c_j the associated shifting distance along p', c_i the shifting distance along the global loop, and we consider two cases.

Forward edge. If p' depends on p with distance d and p' follows p in the loop nest, c_i must be chosen such that $d \geq 0$.

Backward edge. If p' depends on p with distance d and p' precedes p in the loop nest, c_i must be chosen such that $d > 0$.

We may compute c_i, taking into account the global loop shifts over outgoing arcs, the distance d, and whether p' is pipelined or not. The global loop shifts and d are the classical retiming variables and parameters. What happens to p' can be modeled easily, as we previously observed in Section 2.2 that shifting along an inner loop by any amount c_j can be compensated by shifting along the global loop by 1.

Therefore software pipelining p' will increase the total pressure over a cycle by at most 1. This constraint can be modeled by decrementing the distance d when p' is pipelined. We are back to a classical retiming problem, from which we deduce that p' can be pipelined if decrementing d does not induce any cycle with negative or null distance in G_p.

A simple recurrence on the number of pipelined phases concludes the proof.

This is only a lower bound, as G_p does not capture whether pipelining a phase results in an intra-phase shifting of statements involved in inter-phase dependences.

3.2 Necessary and Sufficient Condition

Without information about the statements involved as sink and source of phase dependences, one may only assume that pipelining a phase will incur a shifting constraint along the global loop. In this case, the sufficient condition becomes a necessary one, and the previous proof can be extended to show that the number of phases that can be pipelined while merging prologs and epilogs is exactly k_C, as defined by (1).

Conversely, when considering the full dependence graph G, it is possible to constrain the pipelining of individual phases so that to forbid any inner loop shifting on some specific statements (targets of inter-phase dependences). This will allow to further pipeline some phases without impacting retimability of the global loop. We will come back to this extension when describing the complete algorithm.

4 Global Optimization Problem

Based on Theorem 1, we can formalize the software pipelining of multiple inner loops with prolog-epilog merging as a global optimization problem.

4.1 Multidimensional Knapsack Problem

First of all, the causality preservation condition in Theorem 1 needs to be extended to cover the whole phase dependence graph G_p. Indeed, software-pipelining k_C phases for each (simple) cycle C may create a retiming conflict, as a phase may belong to several cycles and can be chosen to be software-pipelined for one cycle and not for another.

The point is not to pipeline exactly k_C phases in each cycle C, but to minimize the cycle count of the global loop. Since it is not possible to pipeline more than k_C phases (for a given cycle C), the problem leads to the maximization of an objective function under some constraints. The objective function associated with the (static) cycle count for the loop nest is the sum over all phases p of $profit_p = seqtime_p - m_p II_p$, where $seqtime_p$ is the number of cycles to execute phase p and II_p is the initiation interval for the pipelined version of phase p. Let $belongs_{p,C} \in \{0,1\}$ denote whether phase p belongs to cycle C, and let $X_p \in \{0,1\}$ denote whether phase p can be pipelined. The complete optimization problem is stated in Fig. 6.

This is a multidimensional Knapsack problem, a well known NP-complete problem; unlike the one-dimensional case, there is no known pseudo-polynomial algorithm [20] but some heuristics give good results [22].

$$\begin{cases} \text{Variables:} & \forall p \in \{1,\ldots,nb\_phases\}, X_p \in \{0,1\} \\ \text{Objective:} & \max \sum_{p=1}^{nb\_phases} profit_p \times X_p \\ \text{Constraints:} & \forall C \in \{1,\ldots,nb\_cycles\}, \sum_{p=1}^{nb\_phases} belongs_{p,C} \times X_p \leq k_C \end{cases} \quad (2)$$

Fig. 6. Multidimensional knapsack problem to optimize the pipelining profit

$$\begin{cases} \text{Variables:} & \forall p \in \{1,\ldots,nb\_phases\}, X_p \in \{0,1\} \\ & \forall a \in \{1,\ldots,nb\_arrays\}, \forall p \in \{1,\ldots,nb\_phases\}, R_{a,p} \in \{0,1\} \\ \text{Objective:} & \max \sum_{p=1}^{nb\_phases} profit_p \times X_p \\ \text{Constraints:} & \sum_{a=1}^{nb\_arrays} \sum_{p=1}^{nb\_phases} sizeof_a \times R_{a,p} \leq M \\ & \forall C \in \{1,\ldots,nb\_cycles\}, \\ & \quad \sum_{p=1}^{nb\_phases} belongs_{p,C} \times X_p \leq k_C + nb\_phases \times \sum_{a=1}^{nb\_arrays} \sum_{p \in C \wedge assigned_{a,p}} R_{a,p} \end{cases} \quad (3)$$

Fig. 7. Optimizing the pipelining profit with array renaming

4.2 Dependence Removal

So far, we did not consider the applicability of data dependence removal techniques, like privatization and renaming [8,16]. It is reasonable to assume scalar variables have been renamed through conversion to SSA form [3], as is the case in modern optimizing compilers; this guarantees the absence of output (write-after-write) and anti (write-after-read) dependences on *scalar* variables. The case of arrays requires significant static analysis and code generation effort (copy-in and copy-out), and a memory overhead [8,16]. We are not worried by the overheads of copy-in and copy-out, assuming it is amortized over many iterations of the global loop. In addition, we will not consider privatization as the dependences we try to remove involve *distinct source and sink statements*, where array renaming applies. Nevertheless, since prolog-epilog merging is partly motivated by code size improvements, the memory costs of array renaming must be severely controlled. Our work is driven by embedded applications, and we assume a constant bound M on the total memory available for array renaming. Since our technique guarantees the size of the global loop (kernel) does not increase, it is easy to compute such bound, given the original code size and memory footprint of the global loop, for a given local memory configuration.

Removing a dependence may suppress a cycle, hence yield more pipelinable phases, but it also consumes more memory. Overall, the solution is a compromise between the pipelining profit (indirecly linked with the number of pipelinable phases) and the need to keep the amount of extra memory below M. We can model this tradeoff as an extension to the previous linear optimization problem. When renaming a left-hand side (LHS) occurrence of an array, incoming anti-dependences and both incoming and outgoing output dependences to that statement are removed.

We model the decision of renaming an array a in *all* LHS occurrences of instructions of a phase p through a variable $R_{a,p} \in \{0,1\}$. Such variables are

multiplied to "big" constants, controlling which constraints should be nullified—depending on which cycle is broken through array renaming. To capture the correlation between the decision to rename an array and the removal of an inter-phase dependence, it is important that the inter-phase dependence graph G_p is a *multi-graph*: each distinct inter-phase arc in G must yield a distinct arc in G_p. The complete optimization problem is stated in Fig. 7.

nb_arrays denotes the number of arrays, and $sizeof_a$ denotes the size of array a, i.e., the memory overhead of renaming one LHS occurrence, $assigned_{a,p}$ states that array a is assigned in p. The "big" constant is *nb_phases*: it is multiplied by the sum of all variables associated with renaming of some array a in some phase p belonging to a given cycle C. This constant is big enough that the constraint on a cycle C will be nullified *if and only if* one or more renaming occurs along the cycle.

4.3 Prolog-Epilog Merging Algorithm

We may now outline the main steps of the algorithm, assuming a loop nest with multiple phases enclosed by a single global loop. In this section, we focus on solving our optimization problem without considering the impact on downstream loop nest generation methods.

1. If $k_C \geq$ *nb_phases* for every cycle then software-pipeline each phase independently.
2. Otherwise:
   - solve the integer linear optimization problem to identify which are the k_C phases to pipeline;
   - retime the global outer loop, considering phase dependences in G_p, reducing their distance by one every-time the sink phase has been pipelined and contains intra-phase shifted statements at the sink of an inter-phase dependence; this step is guaranteed to terminate according to Theorem 1.
3. Pipeline all remaining phases with the additional constraint that any statement at the *sink* of an inter-phase dependence may not be shifted; in a modulo scheduling algorithm, this constraint can be modeled by forcing such statements to be assigned to column 0 [24]. This step is guaranteed not impact global retiming constraints.
4. Generate the kernel, prolog and epilog of the retimed *global* loop.
5. Generate code for the kernel, prolog and epilog of every pipelined *phase*.
6. Gather all prologs, hoist them *before* the global loop, *after* the prolog of the retimed global loop, and execute them in the same order as phases in the global loop.
7. Gather all epilogs, hoist them *after* the global loop, *before* the prolog of the retimed global loop, and execute them in the same order as phases in the global loop.

4.4 Code Generation

The previous algorithms yield multidimensional shifts resulting from phase pipelining and global loop retiming. However, unlike code generation for single-dimensional pipelining [25], we are not dealing with multiple repetitive patterns in the phase kernels and can rely on the classical code generation methods [24].

Code generation for the retimed global loop only involves classical loop peeling and induction variable substitutions of multidimensional loop shifting [4].

Code generation for pipelined phases after prolog-epilog merging is almost identical to the inner loop pipelining case, except for the following steps.

1. As the loop kernel is now collapsed with the merged prolog and epilog, the trip count of a pipelined phase is *not* decreased by the pipeline depth from the original trip count.
2. When the loop counter occurs in an expression of some shifted statement, one needs to generate an extra induction variable and schedule an extra integer addition in the kernel. In our case, if the statement is shifted by k iterations, the extra induction variable needs to *wrap-around* [11] before proceeding with the last k iterations of the phase. This requires an additional comparison and a conditional move (or a simple mask in case of power-of-two trip counts). These instructions are off the critical path and are not expected to have a high overhead, except on small loop bodies. We will come back to the evaluation of this overhead in the experimental section.

5 Back to the Running Example

Fig. 3 showed how to software pipeline all phases independently. This allows to compute the initiation interval II_p for every phase p. The profit of pipelining a phase is the difference in (static) execution cycles between executing the original inner loop body and the pipelined version. Fig. 8 shows the profit for all phases in the running example, assuming the trip counts of all phases are identical and equal to $m = m_1 = \cdots = m_6$.

The graph G_p was given in Fig. 1. It consists of two cycles, $(ABCD)$ and (CEF). These cycles share phase C, which makes the optimization problem even more interesting as a naive approach may select C to be pipelined for one cycle but not for the other. Fig. 9 shows k_C, the maximum number of phases that can be pipelined for each cycle.

Overall, we face the optimization problem in Fig. 10. A greedy approximation of the solution orders phases from the most profitable to the less profitable, selecting as many phases as possible for software pipelining. A posssible result for the running example is to pipeline A, C, and E, with a total profit of $4m$.

The multidimensional knapsack solution is better: phases A, B, E, F are pipelined, with a total profit of $5m$. Fig. 11 shows the modified phase dependence graph, where pipelined phases are shaded, and the decremented distances of incoming arcs appear in a bold face — following the retiming model of the proof of Theorem 1. Notice phase C is more profitable than B, but pipelining B instead gives us a chance to choose another phase for the other cycle and increases the total profit. This corresponds to a speedup of $13/(13-5) = \mathbf{1.625}$.

As this example shows, it may be overall more effective to pipeline less profitable phases but maximize the overall profit. This observation is very natural when the phases have a different trip count, but our running example shows that this may also occur when cycles in the phase dependence graph are not disjoint.

Phase	A	B	C	D	E	F
Profit	$2m$	$1m$	$1m$	0	$1m$	$1m$

Fig. 8. Profit table

Cycle	$ABCD$	CEF
k_C	2	2

Fig. 9. Constraints

$$\begin{cases} X_j \in 0,1 \\ \max(2X_1 + X_2 + X_3 + X_5 + X_6) \\ X_1 + X_2 + X_3 + X_4 \leq 2 \\ X_3 + X_4 + X_5 + X_6 \leq 2 \end{cases}$$

Fig. 10. Linear program

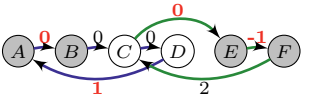


Fig. 11. Phase dependence graph after pipelining A, B, E and F

Fig. 5 shows the final code, with prolog-epilog merging and generation of the outermost prolog and epilog.

6 Related Work and Challenges

We do not aim at extending software pipelining to nested loops, unlike Muthukumar and Doshi [19], Petkov et al. [21], Rong et al. [26,25] and earlier work on multidimensional pipelining (see e.g. Ramanujam et al. [23]). We simply leverage the enclosing loop nest to amortize the fill/flush overhead associated with software pipelining, and to control the code expansion in inner loops.

Compared to plain shifting of statement iterations, our technique involves a more complex combination of affine scheduling [9] and iteration domain splitting (a.k.a. index-set splitting) [10]. This raises many issues, some of which are discussed below.

6.1 Managing Register Pressure

There is an unfortunate side-effect of retiming a prolog (resp. epilog) along the global loop: any live variable entering (resp. leaving) the pipelined kernel will interfere with *every variable in other phases*. The effect on register pressure can be disastrous [27]. There are multiple ways to tackle with this problem.

- The increased pressure is comparable to aggressive scheduling of unrolled or fused loops [17,2]. This should be encouraging given the practical importance of loop fusion among loop optimizations for memory locality and ILP enhancement.
- Spills resulting from inter-phase liveness can always be spilled outside phases. This may turn out to be cheaper than executing the low-ILP prolog/epilog of a deeply pipelined inner loop. It is even more likely to be shorter, especially on architectures with instruction set support for register spill/refill: register stack engine on the Itanium [18], register windows on Sparc, or multi-push/multi-pop operations on CISC instruction sets.

6.2 Managing Code Size

Our method results in code growth outside the global loop only. This is nicer to memory-constrained architectures, but it may still increase cache pollution (or code-copying on local memories). Furthermore, code growth is amplified by the global loop retiming induced by prolog-epilog merging. For innermost loops, *prolog and epilog collapsing* is an alternative strategy consisting in guarding the phases with rotating predicate registers [6,7]. This does not reduce pipeline fill/flush delays however. In our case, pipeline depth has negligible influence on execution time since prologs/epilogs are hoisted outside the global loop.

Muthukumar and Doshi extended the technique to multidimensional software pipelining [19]. They do not target code size reduction, but increased throughput on perfectly nested kernels with low-trip-count innermost loops. Iterations corresponding to prologs and epilogs are shifted over the entire execution of the innermost loop kernel, effectively overlapping iterations of an epilog with those of the next prolog. Compared to prolog-epilog merging, collapsing is difficult to generalize to imperfectly nested loops and incurs harder legality constraints. It is also limited to ISAs with rotating predicate registers. Experimental results on a prototype implementation inside Intel's production compiler are encouraging (despite register pressure challenges similar to ours); this motivates revisiting Muthukumar and Doshi's technique [19] in the context of prolog-epilog merging.

6.3 Multidimensional Scheduling

There are clear opportunities for integrating our technique with other forms of multi-level pipelining, or combined pipelining and parallelization, fusion, etc. [23,2,21,26]. E.g., considering phase C of the running example, it is possible to improve ILP by shifting c_1 by one iteration of the *global* loop. However our approach is not limited to cases where the outer loop is parallel or where unroll-and-jam is legal.

High-level loop optimizations are also promising application of prolog-epilog merging. The polyhedral model is an expressive way to define and search for complex sequences of loop transformations [12]. Yet such complex transformations often induce code size expansion. One source of code duplication comes from multidimensional shifts [28]. It seems possible to integrate our technique in the code-generation phass of a polyhedral compilation tool [12].

7 Experiments

We studied common media and signal-processing applications, including GNU radio, 802.11a (from Nokia), and polyphase image upscaling (from Philips Research).[5] Combining preliminary transformations including inlining, loop rerolling, fusion and if-conversion [1], we could find many occurrences of the "global loop with nested phases" pattern. All these applications exhibit low-trip-count phases, reinforcing

[5] Three benchmarks studied in the ACOTES and SARC FP6 European projects: `http://www.hitech-projects.com/euprojects/ACOTES`, `http://www.sarc-ip.org`

the motivation for prolog-epilog merging. This pattern is also found in many scientific codes; since we target embedded systems, we only studied one of those: the computationally intensive part of the 172.mgrid SPEC CPU2000fp benchmark.

Further experiments are conducted semi-automatically or by hand. This evaluation allows to study the interplay between our technique and other optimizations. It is a required step before undertaking a large integration effort into a back-end optimizer. Figure 12 provides basic statistics about the four applications we studied. It demonstrates the widespread occurrence of loop nests amenable to prolog-epilog merging. The varying trip-counts across neighboring phases (often due to data-dependent control) indicates that loop fusion is not generally applicable. We also verified the presence of many dependence cycles, at all depths, in the four benchmarks; all such cycles contain output/anti-dependences. This confirms the relevance of our global optimization problem with array renaming.

Benchmark	Lines of code	Phases at depth...					Dependences			≠ trip counts
		1	2	3	4	5	Flow	Anti	Output	across phases
GNU radio	427	10	n.a.	n.a.	n.a.	n.a.	3	3	0	100%
802.11a	1502	16	n.a.	n.a.	n.a.	n.a.	5	5	2	50%
Upscaling	150	16	n.a.	n.a.	n.a.	n.a.	17	17	10	25%
172.mgrid	502	5	3	20	17	4	37	37	92	100%

Fig. 12. Applicability of prolog-epilog merging

The rest of the section presents our first results on the polyphase image upscaling benchmark. This code iterates on SD (720×480) YUV video frames and interpolates pixels to double the resolution in both dimensions (1440×960). It accesses a $N^2 \times 512$ lookup-table and two N^2 temporary arrays to iteratively apply filtering, interpolation, and image-enhancement steps over $N \times N$ blocks of pixels. Most time is spent in three-dimensional, imperfectly nested loops, spanning over 150 lines of C code whose control-flow skeleton is depicted in Fig. 13. The 16 phases are labeled A to P. Most of them have N^2 iterations except a couple with $N^2 - 1$. The value of N can be as low as 2 for low-quality interpolation and can grow beyond 5 for very high quality filtering. The default value is $N = 3$ (a typical 3×3 stencil).

All 16 phases can be pipelined (independently of prolog-epilog merging): the dependence graph for this kernel features *some intra-phase loop-carried* dependences but those are associated with reductions and do not hamper pipelining.

```
for(ln = 0; ln < iheight; ln++) {
  for(px = 0; px < iwidth; px++) {
    //// Phase A
    for(index = 0; index < N*N; index++) { ... }
    ...
    //// Phase P
    for(index = 0; index < N*N; index++) { ... }
  }
}
```

Fig. 13. Skeleton of the interpolation code

Memory	Profit
0	918
16	918
36	1107
72	1161
108	1215
144	1269

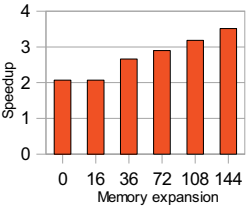


Fig. 14. Trading memory for performance

There are no flow (read-after-write) *inter-phase* dependences, but many output and anti-dependences on the two N^2 temporary arrays; those dependences can be removed through array renaming. The strongly connected components are $\{A, B, C\}$, $\{D, E, H, K, N\}$, $\{F, G\}$, $\{I, J\}$, $\{L, M\}$, $\{O, P\}$.

To break all dependence cycles in the inter-phase dependence graph, the maximal renaming cost is $4 \times N^2 \times 4$ bytes: 4 dependences removed through the renaming of arrays of N^2 32 bit integers. This is very little, both w.r.t. the size of most local memories or L2 caches, and w.r.t. the code size itself. It suggests that array renaming may be a practical solution to allow to pipeline most phases while maintaining the memory overhead close to zero.

Nevertheless, considering the default value $N = 3$, we evaluated the impact of array renaming, varying the upper bound on memory expansion from 0 to $16N^2 = 144$ bytes. The static cycle count for one iteration of the global loop is 1773; the linear optimization problem yields the profit (in static cycles) and speedup figures in Fig. 14. This experiment exhibits 5 steps where extra memory expansion translated into effective improvement of the total profit. It confirms the soundness and relevance of the array renaming for pipelining tradeoff, but more benchmarks should be studied before lessons about the analytical properties of this tradeoff can be learnt.

The next experiment we conducted concerns the interplay of our technique with loop nest optimizers. We studied the behavior of ICC 10.1, the state-of-the-art optimizing compiler from Intel, targeting the Itanium 2 processor (Madison) 1.3 GHz. Among the high-level loop transformations, ICC can perform loop tiling, unroll-and-jam and loop fusion. Only the latter is relevant for this streaming code with little temporal locality. The optimization log shows that 3 pairs of phases are fused, the only relevant ones (in terms of performance) being phases A and B. ICC fails to fuse two phases because of mismatching loop trip counts $(N^2 - 1)$, and it fails to fuse phases L to P due to non-uniform or misaligned dependences. After fusion, 13 phases remain to exercise our technique. This example first shows that our technique is interesting as a *complement* to loop fusion: phases A and B could be fused yet still exhibit opportunities for pipelining and prolog-epilog merging; in addition, our technique is applicable in cases where loop fusion is not.

The last set of experiments aim to evaluate the overheads of prolog-epilog merging w.r.t. plain inner loop pipelining. Those overheads correspond to the extra instruction to compute shifted index variables (see Section 4.4) and to the register pressure induced by live inter-phase variables (see Section 6.1). We considered multiple architecture-compiler pairs: Intel Core 2 Duo 2.4 GHz and Intel Itanium 2 (Madison) 1.3 GHz with GCC 4.3 and ICC 10.1, IBM Cell PowerPC 3.2 GHz with GCC 4.1, STMicroelectronics ST231 400 MHz (embedded VLIW, 4-issue) with st200cc 1.9.0B (Open64). We used -O3 optimization, with pipelining turned off, with and without loop unrolling, and manually pipelined the most significant phases (source-level). In all cases, prolog-epilog merging performed better than unpipelined code, and sometimes even better than plain pipelining (phase L with GCC). We also verified that lower values of N improve

the benefit of prolog-epilog merging: no iteration is spent on fill/flush except at the very beginning/end of the global loop [19]. This preliminary experiment confirms that the intrinsic overheads of our technique can be amortized.

8 Conclusion

Software pipelining with prolog-epilog merging may appear as the most natural extension to inner loop pipelining. Indeed, it avoids the code size and execution time overhead of nested prologs and epilogs: these advantages over loop unrolling are exactly the motivations that drove to the deign of software pipelining algorithms [14]. We formalized the concept of prolog-epilog merging, combining inner loop pipelining with multidimensional retiming. We combined our technique with array renaming to improve the pipelinability of inner loops. This results in a global scheduling and memory expansion tradeoff, modeled as a tractable, integer linear optimization problem.

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufman, San Francisco (2002)
2. Carr, S., Ding, C., Sweany, P.: Improving software pipelining with unroll-and-jam. In: Proceedings of the 29th Hawaii Intl. Conf. on System Sciences (HICSS 1996). Software Technology and Architecture, vol. 1. IEEE, Los Alamitos (1996)
3. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. on Programming Languages and Systems 13(4), 451–490 (1991)
4. Darte, A., Huard, G.: Loop shifting for loop parallelization. Intl. J. of Parallel Programming 28(5), 499–534 (2000)
5. Darte, A., Silber, G.-A., Vivien, F.: Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. Parallel Processing Letters 7(4), 379–392 (1997)
6. Dehnert, J.C., Hsu, P.Y., Bratt, J.P.: Overlapped loop support in the Cydra 5. In: Intl Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1989), pp. 26–38 (April 1989)
7. Dulong, C., Krishnaiyer, R., Kulkarni, D., Lavery, D., Li, W., Ng, J., Sehr, D.: An overview of the Intel IA-64 compiler. Intel. Technical Journal Q4 (1999)
8. Feautrier, P.: Array expansion. In: Intl. Conf. on Supercomputing (ICS 1988), St. Malo, France, pp. 429–441 (July 1988)
9. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part I, multidimensional time. Intl. J. of Parallel Programming 21(6), 315–348 (1992)
10. Feautrier, P., Griebl, M., Lengauer, C.: On index set splitting. In: Parallel Architectures and Compilation Techniques (PACT 1999). IEEE Computer Society, Los Alamitos (1999)
11. Gerlek, M.P., Stoltz, E., Wolfe, M.J.: Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. ACM Trans. on Programming Languages and Systems 17(1), 85–122 (1995)

12. Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., Temam, O.: Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. Intl. J. of Parallel Programming, 57 (2006); Special issue on Microgrids
13. Karczmarek, M., Thies, W., Amarasinghe, S.: Phased scheduling of stream programs. In: LCTES 2003 (June 2003)
14. Lam, M.S.: Software pipelining: An effective scheduling technique for vliw machines. In: ACM Principles, Logics, and Implementations of High-Level Programming Languages (1988)
15. Leiserson, C.E., Saxe, J.B.: Retiming synchronous circuitry. Algorithmica 6(1), 5–35 (1991)
16. Maydan, D.E., Amarasinghe, S.P., Lam, M.S.: Array dataflow analysis and its use in array privatization. In: Principles of Programming Languages (PoPL 1993), Charleston, South Carolina, pp. 2–15 (January 1993)
17. McKinley, K., Carr, S., Tseng, C.-W.: Improving data locality with loop transformations. ACM Transactions on Programming Languages and Systems 18(4), 424–453 (1996)
18. McNairy, C., Soltis, D.: Itanium 2 processor microarchitecture. IEEE Micro., 44–55 (March 2003)
19. Muthukumar, K., Doshi, G.: Software pipelining of nested loops. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027. Springer, Heidelberg (2001)
20. Parra-Hermandez, R., Dimopoulos, N.J.: A new heuristic for solving the multichoice multidimensional knapsack problem. IEEE Transactions on Systems, Man, and Cybernetics — Part A: Systems and Humans 35(5) (September 2005)
21. Petkov, D., Harr, R.E., Amarasinghe, S.P.: Efficient pipelining of nested loops: Unroll-and-squash. In: Proc. of the 16th Intl. Parallel and Distributed Processing Symp. (IPDPS 2002), Washington, DC (2002)
22. Puchinger, J., Raidl, G.R., Pfershy, U.: The multidimensional knapsack problem: Structure and algorithms. Technical Report No. 006149 INFORMS Journal of Computing (March 2007)
23. Ramanujam, J.: Optimal software pipelining of nested loops. In: International Symposium on Parallel Processing, Washington, D.C, pp. 335–342 (1994)
24. Rau, B.R.: Iterative modulo scheduling: an algorithm for software pipelining loops. In: MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture, pp. 63–74. ACM Press, New York (1994)
25. Rong, H., Tang, Z., Govindarajan, R., Douillet, A., Gao, G.R.: Code generation for single-dimension software pipelining for multi-dimensional loops. In: Proceedings of the International Symposium on Code generation and Optimization(CGO 2004), pp. 175–186 (March 2004)
26. Rong, H., Tang, Z., Govindarajan, R., Douillet, A., Gao, G.R.: Single-dimension software pipelining for multi-dimensional loops. In: Proceedings of the International Symposium on Code generation and Optimization(CGO 2004), pp. 163–184 (2004)
27. Touati, S., Eisenbeis, C.: Early Control of Register Pressure for Software Pipelined Loops. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 17–32. Springer, Heidelberg (2003)
28. Vasilache, N., Cohen, A., Pouchet, L.-N.: Automatic correction of loop transformations. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671. Springer, Heidelberg (2007)
29. Verdoolaege, S., Bruynooghe, M., Janssens, G., Catthoor, F.: Multi-dimentsional incremetal loops fusion for data locality. In: ASAP, pp. 17–27 (2003)

A Flexible Code Compression Scheme Using Partitioned Look-Up Tables

Martin Thuresson, Magnus Själander, and Per Stenstrom

Department of Computer Science and Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden

Abstract. Wide instruction formats make it possible to control microarchitecture resources more precisely by the compiler by either enabling more parallelism (VLIW) or by saving power. Unfortunately, wide instructions impose a high pressure on the memory system due to an increased instruction-fetch bandwidth and a larger code working set/footprint.

This paper presents a code compression scheme that allows the compiler to select what subset of a wide instruction set to use in each program phase at the granularity of basic blocks based on a profiling methodology. The decompression engine comprises a set of tables that convert a narrow instruction into a wide instruction in a dynamic fashion. The paper also presents a method for how to configure and dimension the decompression engine and how to generate a compressed program with embedded instructions that dynamically manage the tables in the decompression engine.

We find that the 77 control bits in the original FlexCore instruction format can be reduced to 32 bits offering a compression of 58% and a modest performance overhead of less than 1% for management of the decompression tables.

1 Introduction

Traditional RISC-like Instruction-Set-Architectures (ISAs) offer a fairly compact coding of instructions that preserves precious instruction-fetch bandwidth and also makes good use of memory resources. However, densely coded instructions tend to increase the efficiency gap between general-purpose processors (GPPs) and tailor-made electronic devices (ASICs) by not being capable of finely controlling microarchitecture resources. In fact, with the advances in compiler technology it is interesting to let wider instructions expose a finer-grain control to the compiler.

Very-Long-Instruction-Word (VLIW) ISAs do exactly this by exploiting parallelism across functional units, whereas architectures with exposed control such as NISC [1] and FlexCore [2] do it in order to expose the entire control to the compiler, thereby having a potential to reduce the efficiency gap between GPPs and ASICs. In fact, recent VLIW ISAs such as IA-64 [3] use 128-bit instruction

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 95–109, 2009.

bundles containing three instructions each and FlexCore uses as many as 109 bits per instruction.

The downside of wider instructions, however, manifests itself in at least three ways: a higher instruction-fetch bandwidth, a larger instruction working-set, and a larger static code size. This may lead to higher power/energy consumption as well as lower performance, which may in fact outweigh the gains of more efficient use of microarchitecture resources.

Previous approaches to maintain the full expressiveness of wide instruction formats and yet reducing the pressure on the memory system have been to code frequently-used wide instructions more densely. Mips16 [4] and ARM Thumb [5] provide a more dense alternative instruction set and it is possible to switch between the wide and dense instruction formats. In another approach, a dictionary is provided that expands a densely coded instruction into a wide instruction either by coding a single wide instruction with a denser codeword [6,7] or by coding a sequence of recurring wide instructions with a denser codeword [8,9,10,11]. Regardless of the approach, the drawback of all these schemes is that they can only utilize a fraction of the expressiveness of the wide instruction format either because only a subset is compressed or because of the huge dictionaries needed, which can incur significant run-time costs. Our aim is a more scalable approach that can accommodate large programs.

This paper contributes with a novel code-compression scheme that utilizes the *full* expressiveness of the wide instructions by coding the program in a dense fashion. The decompression engine comprises a set of look-up tables (LUTs), each used to compress a partition of the wide instruction word. The compression is done off-line at compile-time by analyzing what subset of the wide instruction set is used in each basic block through a profiling pass. We present an algorithm for compression of the program using dense instructions and for management of the decompression engine at run-time by changing the dictionary entries on-the-fly and yet keeping the run-time costs low. The end result is a decompression methodology that can utilize the full expressiveness of the wide instruction format with low run-time costs. The paper also presents a methodology for how to configure and dimension the decompression engine under various constraints such as keeping the latency of LUTs at a low level.

Based on the FlexCore [2] architecture, we show that the original 77-bit instruction word can be reduced by 58% with less than 1% percent run-time cost in the number of executed instructions for manipulating the LUTs for a set of media benchmarks from the EEMBC suite [12].

As for the rest of the paper, Section 2 describes our baseline architecture model, the FlexCore architecture, followed by a description of the new compression scheme in Section 3. In Section 4, a method for selection of the configuration of the LUTs is presented, and in Section 5 an algorithm for generating the compressed program is shown. The experimental methodology and results are presented in Sections 6 and 7, respectively. Related works are discussed in Section 8 and the paper is concluded in Section 9.

2 FlexCore

FlexCore [2] is an architecture with exposed control, based on the functional units found in a typical five-stage general-purpose pipeline. The data-path consists of a register file, an arithmetic-logic unit (ALU), a multiplier, a load/store unit and a program-control unit connected to each other using a fully connected interconnect and controlled using a wide control word. Figure 1 shows an illustration of the architecture, with the control on top, and the interconnect at the bottom of the figure. One unique property of the FlexCore architecture is that it is possible to include hardware accelerators in the framework and use the interconnect and the general control to flexibly configure a pipeline out of the available datapath elements. Another novel aspect of FlexCore is that its control space is a superset of a traditional five-stage general-purpose processor (GPP),

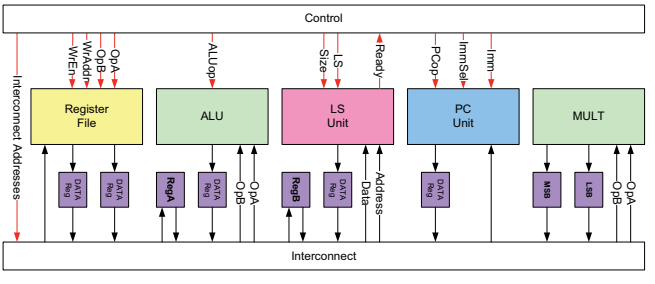


Fig. 1. FlexCore, an architecture using a wide control word, a fully connected interconnect, and the datapath units found in a typical early five-stage load/store architecture such as MIPS 2000

Table 1. Control signals in the FlexCore architecture. Size given in number of bits.

Signal name	Description	Size	Signal name	Description	Size
RegReadA	Reg. A read address	5	PCOp	PC operation	3
RegAStall	Reg. A read stall	1	PCStall	PC stall	1
RegReadB	Reg. B read address	5	I_ALUA	Inter. ALU A	4
RegBStall	Reg. B read stall	1	I_ALUB	Inter. ALU B	4
RegWrite	Reg. write address	5	I_RegWrite	Inter. Reg write	4
RegWE	Reg. write-enable	1	I_LSWrite	Inter. L/S Write	4
Buf1	Buf1 write-enable	1	I_LS	Inter. L/S address	4
Buf2	Buf2 write-enable	1	I_Buf1	Inter. buf 1	4
ALUOp	ALU operation	4	I_Buf2	Inter. buf 2	4
ALUStall	ALU stall	1	I_CtrlFB	Inter. ctrl feedback	4
LSOp	L/S operation	2	MultStall	Mult stall	1
LSSize	L/S size	2	MultEnable	Mult enable	1
LSStall	L/S stall	1	I_MultA	Inter. mult A	4
PC	PC immediate select	1	I_MultB	Inter. mult B	4

making it possible to fall back on traditionally scheduled instructions found in load/store architectures such as MIPS R2000, if needed.

While this architecture has been shown to be more efficient in terms of execution time and cycle count for embedded benchmarks than a five-stage single-issue pipelined GPP counterpart, the cost in instruction-fetch bandwidth is three times higher compared to a traditional GPP like MIPS R2000, and almost as much in terms of static code size [2]. This makes FlexCore a suitable target architecture for code compression schemes, especially since embedded devices usually have very tight constraints on memory usage and power/energy consumption.

The full control word for the FlexSoC, which consists of 109 signals (out of which 32 comprise the immediate values), can be seen in Table 1. So 77 of the 109 bits are for control, which is the target in this study.

3 The Instruction Compression Scheme: Overview

The compression scheme leverages on the fact that during phases of the execution, some combinations of control bits will never appear in the instruction stream. Since the expressiveness found in the wide instructions is thus not utilized, a more efficient encoding scheme can be used. The encoding scheme uses look-up tables (LUTs) to store bit patterns and the compressed instruction is a list of indexes into these tables. The bits found in the tables are then merged to form the decompressed instruction, which can then be executed. Figure 2 shows a decompression structure with four LUTs that together generate the wide instruction. Because of the simple logic involved, and relatively small LUTs needed, we will later show that decompression can be done with virtually no performance overhead as part of the instruction fetch.

The contents of the LUTs can be changed using dedicated *table-manipulating instructions* in the instruction stream. This allows the compiler to use small tables, whose contents are tuned for the particular phase of the execution. The placement of these dedicated instructions will affect the quality of the final solution. The static number of table-manipulating instructions will affect the static code size, whereas the number of table-manipulating instructions in the dynamic

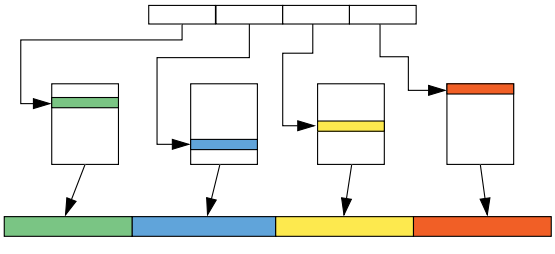


Fig. 2. The instruction-decompression structure encompassing a number of LUTs that expand different parts of the narrow instruction into the wide complete control word

instruction stream will affect the performance overhead in terms of instruction overhead and potentially more instruction-cache misses.

In this study, we adopt a straight-forward strategy – a single table-manipulating instruction, called a *LUT-load*, updates a single entry in one look-up table. While this makes our results pessimistic in terms of the overhead caused by the table-manipulating instructions, we note that this is an area for improvement that is subject for future research.

An important design trade-off in this scheme is the size and configuration of the LUTs. The instruction size depends on both the number of tables, and the size of each. The number of tables dictates the number of indices in the narrow instruction format, and the number of entries dictates the number of bits needed for each index. Also, the size of the tables will influence how often the contents of the tables need to be updated. In the next section, we present a methodology for dimensioning the decompression engine taking this into account.

The methodology also takes into account that many bits in the control word are so called "don't-care" values meaning that they can be set to either zero or one, without affecting the correctness of the program. Don't-care signals have previously been used successfully in the NISC project [6]. The FlexCore-compiler has been updated to generate programs where the "bits" can be 0, 1 or X, giving the compression algorithm additional opportunities for optimizations.

4 A Method for Wide-Instruction Partitioning

In this section, we present a methodology to dimension the compression-engine tables with respect to the number of tables, the number of entries in each table, and the partitioning of the wide instructions across the tables. Since the tables are fixed in the architecture, this design decision needs to be done before fabrication, and is thus done only once.

The method consists of four steps, each one illustrated with examples using the FlexCore control word. In the first step, the designer identifies bits in the control word that are highly correlated and should always be placed in the same LUT. These sets of bits are called *sub-groups*. Table 2 lists the sub-groups identified in the FlexCore architecture.

In the next step, all possible subsets of the set of sub-groups are generated to create possible LUT candidates. A *candidate* is a set of bits that together could become a LUT in the design. An optimization is to already here remove candidates which will not be in the final solution; for example if they are too narrow or too wide. In FlexCore, for example, we might decide to only consider candidates with a width between 7 and 16 bits because of delay and power constraints. Here (RegA, RegB) is a candidate, but (Buf, I_Buf) is too narrow to be a reasonable one, since too many small LUTs lead to a lower degree of compression.

In the third step, LUT candidates are combined into groups called *possible solutions*, so that the candidates in the group cover all the bits in the control word once, and only once. One of many possible solutions in our example is the following

Table 2. Sub-groups for the FlexCore architecture. Size given in number of bits.

Sub-group	Signals included	Size	Sub-group	Signals included	Size
RegA	RegReadA + RegAStall	6	Buf	Buf1 + Buf2	2
RegB	RegReadB + RegBStall	6	I_ALUA	I_ALUA	4
RegW	RegWrite + RegWE	6	I_ALUB	I_ALUB	4
PC	PC + PCOp + PCStall	5	I_RegW	I_RegWrite	4
ALU	ALUOp + AluStall	5	I_LS	I_LSWrite + I_LS	8
LS	LSOp + LSSize + LSStall	5	I_Buf1	I_Buf1	4
Mult	MultStall + MultEnable I_MultA + I_MultB	10	I_Buf2	I_Buf2	4
			I_CtrlFB	I_CtrlFB	4

candidate list: (I_RegW, I_LS, RegW), (I_ALUB, I_Buf2), (I_ALUA, Mult), (LS, PC), (ALU, I_CtrlFB Buf, I_Buf1) (RegB, RegA).

Finally, each of the possible solutions is evaluated using a user-defined cost function. The cost function evaluates how good the possible solution is for a given application (called workload), and returns a numerical result (lower is better). This makes it possible to find a solution that is relevant for the type of applications that will be executed on the system. In our experiments, we have used cost functions for LUT-access time, compressed instruction width, and energy efficiency. Several cost functions can be given with different priority, and only if a high priority function ties, a lower is evaluated. This makes it easy to add hard design-constraints, such as a maximum access time for the tables, and to make sure that the design fits within a given power envelope.

The cost function will take the LUT configuration into account given by the possible solution, and calculate the LUT sizes needed for the workload, so the whole program can be executed without inserting LUT-loads. In the LUT size calculation, don't-care bits are greedily set to 0 or 1 if it makes it possible to merge two entries into the same value. For example, if the LUT already holds the entry 1X00X, and we try to add the entry X100X, the LUT would be updated to hold the entry 1100X. With this information, the cost can be calculated and returned. Once all the possible solutions have been evaluated, the designer is presented with a list of the solutions with the lowest cost.

While each run of the algorithm gives a list of solutions tuned for that particular workload, we propose running it several times with different workloads. Among all the saved solutions, the designer can pick one that works well for all of the workloads.

One challenge in this methodology is to find a suitable workload. The workload should, for the best solutions, produce LUTs that have a size that results in an acceptable access time and power consumption. For the benchmarks used in this paper, we selected a subset of the full program by running the EEMBC benchmarks one iteration (using the flag *-i1*) and all the program counter-values for the executed instructions were recorded. For each benchmark, the footprint used was the subset of instructions selected by the program-counter trace.

The algorithm also reports the size needed for each suggested LUT. Since the compressed program can change the contents of each LUT, the distribution of the sizes can be used to determine the final sizes for the LUTs.

5 Algorithm for Generation of a Compressed Program

In order to execute a program on a system using our proposed compression scheme, the generated binaries need to be updated by inserting the LUT-load instructions inside the binaries. We have developed an algorithm for generating a compressed program with the LUT-loads placed to keep the performance penalty low. One important approach is to avoid placing LUT-loads in basic blocks that are frequently executed, such as inside inner loops. Since the original wide instructions are compressed, it is possible to reduce the instruction footprint size.

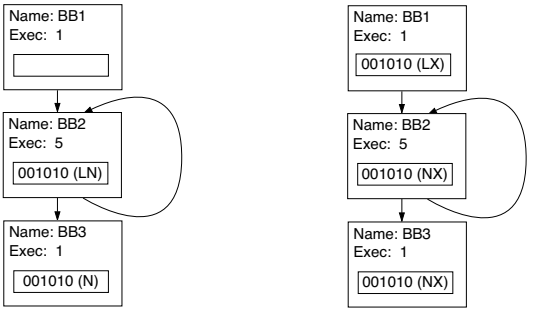
A profiling run is used to get the execution frequency for the basic blocks. We can then estimate the performance overhead caused by the LUT-loads simply by multiplying the number of required table manipulations in each basic block with that block's execution count.

In order to generate the final compressed program, the algorithm uses the uncompressed program, a compiler-generated flow graph showing the relationship between all basic blocks, profiling information at the basic block level, and the LUT-table configuration.

The algorithm uses the flow graph as its main data structure and it associates a complete list of entries with each basic block that are needed in the tables for the basic block to be executed. An entry in this context means a value that is present in a LUT in this basic block. If any basic block requires more entries than the capacity of the tables, the basic block is split into several basic blocks.

The algorithm uses three flags, while updating the flow-graph. The flag L (Load), is used for any entry that does not exist in *all* of the possible predecessors to a given basic block. It is significant since only entries that are marked with L will actually generate LUT-load instructions. The flag N (Needed) shows that an instruction in the basic block requires the entry to be present in the LUT. These entries can never be removed from the basic block, since the code would not work without it. Finally, the flag X (Locked) is used by the algorithm to tag an entry as processed. This makes it easy to process the entries one-by-one, and make sure that all are visited once, and only once.

Initially, all entries in the flow-graph are scanned, and the L and N flags are set according to the description above. Figure 3(a) shows a flow-graph for a simple loop used to illustrate the algorithm, with each basic block annotated with its execution count from the profiling run. Also, the entry 001010 for one particular LUT is shown with its flags. The entry is initially stored in two different basic blocks, but only marked with L in BB2, since the only path into BB3 is through BB2. The performance penalty for this particular state is five, since a LUT-load in BB2 would be executed five times.



(a) Before optimization

(b) After optimization

Fig. 3. Example of a simple flow-graph used to illustrate the algorithm. The graphs show the locations a particular entry in one of the LUTs before and after BB2 is optimized.

```
// Calculate possible cost achieveble by (recursively) moving
// entry up to all predecessors
optimizeLoad(current_node, entry)
  if(recursionLevel > MAX || current_node.IsRootNode() == 0)
    return INT_MAX
  cost = 0;
  loop over predecessors using pred
    if(pred.HasEntry(entry) cost += 0
    elseif(pred.IsFull())   return INT_MAX // No room to push the entry
    else
      cost_here = pred.ExecutionCount()   // Cost if not pushed further
      cost_push = optimizeLoad(pred, entry)  // Cost if pushed further
      if(cost_here <= cost_push)
         cost += cost_here
      else
         cost += cost_push
         pred.Add(entry) // Add speculatively
  return cost
```

Fig. 4. Simplified pseudo-code for the recursive function. The complete version stores LUT-updates locally and returns the solution.

The next step of the algorithm will process the entries marked with L one-by-one, until all have been processed. The main idea behind the algorithm is to look at one entry at a time to see if it is possible, with a lower cost, to make sure the entry already exists in of the possible predecessors. This is done using a recursive optimization function that works as follows.

For each entry that is marked L in the graph, the optimization function tries to "push" the entry to all the predecessors of the basic block. The pseudo-code in Figure 4 shows on a high-level how the function works. It keeps track of the cost as it pushes the entry to the predecessors and returns the best cost it can. The algorithm continues to recursively push the entry further until one of several conditions occur: 1) the value already exists in a basic block; here don't-care values are greedily resolved to zeros or ones, if it helps to find a match

2) the table in the basic block is full; 3) it reaches a root node in the graph; or 4) a maximum recursion depth has been reached. If the minimum cost found is smaller than the current cost, the new entries are added to the affected basic blocks and all the flags are updated. Being able to push entries marked with L out of loops is essential for getting a well performing solution. Since each optimization may increase the number of entries in the LUTs, the entries in the basic blocks that have the most L-flags are processed first.

To keep the execution time of the algorithm down, the recursive depth should be set. This bounds the computational complexity, which would otherwise be $O(n^2)$, where n is the number of basic blocks. In our experiments, we have found that very little improvement was found for a depth larger than 30, meaning a load is at most pushed 30 basic blocks.

Figure 3(b) shows the resulting graph after the entry in BB2 has been processed by the recursive function. The execution cost overhead associated with this particular entry has now been reduced to one.

Using the graph it is trivial to generate a compressed program, by issuing LUT-loads at the start of each basic block for each entry marked L. Because of the extra instruction, some branch offsets may have to be recalculated.

6 Experimental Methodology

In order to evaluate our scheme, we have applied the table-configuration and program compression algorithms to the FlexCore architecture and evaluated them using the EEMBC [12] benchmarks Autocorr, FFT, and Viterbi. We have only considered LUT candidates with a table width between 5 and 16 and table sizes that are sufficiently fast and energy efficient according to our cost functions to be defined below.

Table 3. Default LUT configuration used as a baseline for comparisons

Included signals	Size (bits)	Included signals	Size (bits)
ALU + I_ALUA + I_ALUB	13	LS + I_LS	13
RegReadA	6	Buf + I_Buf1 + I_Buf2	10
RegReadB	6	PC + I_FB	9
RegWrite + I_RegWrite	10	Mult	10

To compare the selected solution with a baseline solution, we use a configuration where signals that, to the best of our knowledge, fit together are placed in the same table. These groups, referred to as the *default* configuration, can be seen in Table 3. In our case study, we combined several cost functions with different priorities to meet timing, size, and power constraints. Since a lower priority function is only evaluated if there is a tie between solutions using a higher priority cost function, it is important to carefully design the cost functions so that ties actually occur. The cost functions used in this study illustrate how this can be done.

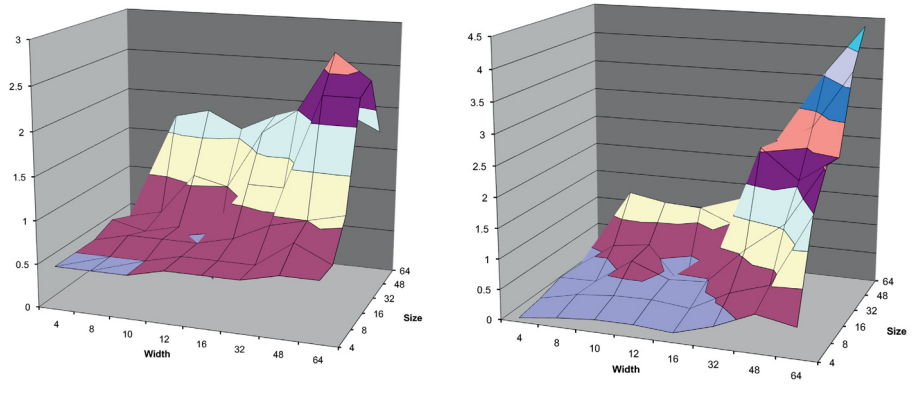
The first cost function makes sure that the access time is below one nanosecond for all LUTs. It is designed as a step function, which returns the value one if the delay is below one nanosecond, and a larger value otherwise. The second cost function makes sure we have a sufficiently small instruction format. By dividing the resulting compressed instruction-size by eight (using integer division), the output is quantized. Finally, ties among the solutions with the smallest instructions are broken using a third cost function, which adds the power values of all the tables in the solution. For all functions, a lower cost is better.

Results on timing and power estimates come from RC-extracted layouts of LUTs of various sizes – depth as well as width have been varied between 4 and 64. The LUTs were described in VHDL and taken through a Cadence Encounter synthesis, placement-and-routing flow for a commercial 65-nm process technology (low-power cell library with standard V_T). The estimates shown in Section 7 were obtained for the worst-case 125 °C corner at 1.1 V.

7 Experimental Results

7.1 Look-Up Table Configuration

The data used for the cost functions for delay and power is shown in Figure 5. The graphs show how the LUT dimensions influence power and timing of the decoder. The delay increases with the number of entries in the LUT. This comes as no surprise since the multiplexers for reading and writing data into and out of the LUT need to be wider, thus increasing the logic depth. However, the width also has a negative effect on timing. This is due to longer wires that span across the larger (wider) LUT, which increases the propagation delay. For the power it is the opposite with the width being the more dominant factor. As the



(a) LUT Timing (b) LUT Power

Fig. 5. Timing and power estimates from RC-extracted 65-nm layouts of LUTs of various sizes

LUT becomes wider there are more signals and logic to switch, since there are more bits to read or write in a single cycle, thus increasing the dynamic power. However, an increased number of entries in a LUT also increases the power. This is due to an increase in logic, which results in a higher static power as well as longer wires that require more power for driving them. From Figure 5 one can see that the power dissipation increases rapidly for LUTs wider than 16 bits. The data motivated us to only look at LUTs that are less than 16 bits wide in our case study.

As described in Section 4, the working sets used to decide on a LUT configuration were created by running the program once and only use the executed instructions. For our benchmarks, only 45% to 60% of the entire set of instructions were executed during such a run. When using this subset of the instructions, our algorithm provided solutions that met our design goals of an access time of less than one nanosecond.

Table 4 shows the delay, number of bits per instruction, and the total power for the default configuration (Table 3) as well as the configuration with the lowest cost found for each of the benchmark subsets. Here we see that the default configuration has a much higher access time than the configuration selected by our algorithm. While the compressed instruction size is similar in the two configurations, the selected configuration is more power efficient.

Table 4. Results for the table configurations that hold the complete program. Results for both the algorithm-selected configuration (best), and the default configuration are shown.

Benchmark	Delay (ns)	Inst. Width (bits)	Power (mW)
Autocorr (best)	0.6	28	3.8
Autocorr (default)	1.7	27	5.0
FFT (best)	0.6	29	4.8
FFT (default)	1.8	28	4.7
Viterbi (best)	0.6	28	3.8
Viterbi (default)	1.8	27	5.3

The next step is to look at the best configuration for each benchmark, and find a one that works overall best, i.e. for all benchmarks. The methodology used was to consider the cost functions for the ten best solutions for each benchmark. Among the possible solutions that had less than one nanosecond latency for all application, we selected the solution that had the narrowest compressed instruction width.

Table 5 shows the selected configuration. For each LUT we also list the width and the total number of entries needed to fit the entire program for each benchmark. With this info, we also list the sizes we use for each table. The sizes were selected to make sure that only 32 bits are needed to index the LUTs. For our configuration, the total number of bits stored in LUTs are only 1036 bits. The final configuration compresses the 77 control signals (109 minus the immediate) down to 32 bits, a compression ratio for each instruction of 58% (41% with the 32-bit immediate included).

Table 5. Required number of entries for each table for each application and the size chosen for our implementation. The selected sizes are chosen to get a total instruction size of 32 bits for the control bits.

Included signals	Group width	Suggested size			Chosen size
		Autocorr	FFT	Viterbi	
I_RegW, I_LS	12	4	5	4	4
I_ALUB, I_Buf2	8	5	6	5	8
I_ALUA, Mult	14	4	8	4	8
LS, PC,	10	15	15	15	16
ALU, I_CtrlFB	9	16	17	16	16
Buf, I_Buf1	6	6	6	6	4
RegW	6	25	28	21	32
RegB	6	18	23	20	16
RegA	6	21	24	20	32

Table 6. Compression results for our benchmarks. Cost is the fraction of executed LUT-loads when running our compressed program. Static code size only includes the control bits targeted by our compression.

Benchmark	Dynamic Instruction Count			Static Code Size		
	Normal Instr.	LUT-Loads	Cost	Uncompressed	Compressed	Compr. Ratio
Autocorr	25096	268	1.1%	16.9kB	15.9kB	6%
FFT	169417	2086	1.2%	17.4kB	16.0kB	9%
Viterbi	293755	519	0.6%	15.9kB	14.0kB	12%

7.2 Program Generation

The next step is to generate code optimized for the selected table-configuration. The benchmarks were compressed using the algorithm outlined above with a maximum recursion level of 30. Table 6 lists the number of "normal" instructions and LUT-loads executed when running the benchmarks. In terms of performance overhead, the results clearly show that the algorithm is successful at placing the LUT-loads. Using the pessimistic calculation that each entry takes one cycle to change, we see that the overhead for changing the contents of the tables is about one percent for all three applications.

Regarding the code size, the static code now contains the compressed instructions *and* the extra table of load instructions. For the three benchmarks, the static code size targeted by our scheme only decreased by between 6% and 12%, as seen in Table 6. The results can be explained by the fact that our algorithm is optimized for the reduction of performance overhead, and not static code size. In fact, very few LUT-loads were placed in the 90% most frequently executed instructions, leading to a smaller instruction working set for these blocks. An interesting expansion of this work would be to consider other optimization goals for the compression algorithm.

8 Related Work

The NISC project proposes the use of one or two look-up tables to efficiently store programs for FPGA-based custom IPs [6]. While they achieve very high compression ratios (3.3 times for their applications), the size of the tables becomes quite large. Using their approach, with one static LUT, the EEMBC benchmarks evaluated in our study would require between 300 to 400 entries each, making it more suitable in an FPGA environment than to be placed in the latency-critical front-end of an ASIC microarchitecture in which the clock frequency would be heavily constrained.

IBM CodePack [7] is a compression method which enables unmodified cores to run compressed instructions. The instructions are encoded using two tables, one for the op-code and one for the operands. The compressed instructions access the tables using variable length code-words. Because of the complexity of the decompression, a cache is required to hide the decompression latency.

Benini et al. [13] increase the working set that can be placed in the instruction cache by storing the N most frequently executed instructions in a table, and replace them in the code with $log_2(N)$ bit wide codewords. Dictionary-based compression has also been used to replace sequences of instructions with one codeword [10,11]. While it is not clear how these approaches scale as the instruction width increases and the compiler gets more opportunities to optimize the control word, it is likely that the wider an instruction is, the less likely it is to be reused.

Brorsson and Collin [14] extend previous work in dictionary-based compression by considering dictionary sharing between applications. Similar to our work, they use an instruction for updating the state of the tables, though they only change the contents of the LUTs during context switches, not between execution phases in the same program.

9 Conclusion and Future Work

This paper presents a novel code compression approach which reduces the pressure on the memory system in wide instruction architectures. Using FlexCore as a case study we show that the scheme, which is based on small look-up tables that each compresses parts of the control word, can compress the 77-bit control-word down to 32 bits with only 1% performance penalty because of the instructions that update the LUTs.

To aid the designer in configuring and dimensioning the look-up tables, a methodology using cost functions is presented. Also, a method to generate compressed programs optimized for a low performance overhead is described.

Accurate profiling information is important for achieving an efficient compressed program. In this study, instruction traces from the execution of the programs were used to get execution counts for each basic block. Another method that could be used to get fast and accurate profiling information is to use sampling techniques developed for feedback-driven compiler optimizations [15].

The compression scheme focuses on compressing the control bits in the instruction word, leaving the immediate field untouched. Frequent value encoding [16] and significance-width compression [17,18,19] are compression approaches, which have been shown to be very effective at compressing memory traffic [20] and could be a promising approach to use here as well, but this is left for future work.

Acknowledgments

The authors thank Lars Svensson for fruitful discussion on the compression algorithm. We also thank Thomas Schilling for his work on the FlexCore tool-chain and all the members of the FlexSoC project for their contribution to the FlexCore architecture.

References

1. Reshadi, M., Gorjiara, B., Gajski, D.: Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths. In: Proceedings of the 23rd International Conference on Computer Design (ICCD), pp. 69–76. IEEE Computer Society, Los Alamitos (2005)
2. Thuresson, M., Själander, M., Björk, M., Svensson, L., Larsson-Edefors, P., Stenstrom, P.: FlexCore: Utilizing exposed datapath control for efficient computing. Journal of Signal Processing Systems (to appear, 2008); Accepted on the 4th of March 2008
3. Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H., Zahir, R.: Introducing the IA-64 architecture. IEEE Micro. 20(5), 12–23 (2000)
4. Kissell, K.: MIPS16: High-density MIPS for the Embedded Market. Silicon Graphics MIPS Group (1997)
5. Advanced RISC Machines Ltd.: An Introduction to THUMB (March 1995)
6. Gorjiara, B., Gajski, D.: FPGA-friendly code compression for horizontal microcoded custom IPs. In: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays (ISFPGA), pp. 108–115. ACM Press, New York (2007)
7. Game, M., Booker, A.: CodePack: Code Compression for PowerPC Processors. International Business Machines (IBM) Corporation (1998)
8. Lefurgy, C.R.: Efficient execution of compressed programs. PhD thesis, Ann Arbor, MI, USA, Chair-Trevor Mudge (2000)
9. Lefurgy, C., Piccininni, E., Mudge, T.N.: Reducing code size with run-time decompression. In: Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA), pp. 218–228. IEEE, Los Alamitos (2000)
10. Corliss, M.L., Lewis, E.C., Roth, A.: DISE: A programmable macro engine for customizing applications. In: Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA), pp. 362–373. ACM Press, New York (2003)
11. Thuresson, M., Stenstrom, P.: Evaluation of extended dictionary-based static code compression schemes. In: Proceedings of the 2nd conference on Computing Frontiers (CF), pp. 77–86. ACM Press, New York (2005)

12. EEMBC, the embedded microprocessor benchmark consortium (2008), http://www.eembc.org
13. Benini, L., Macii, A., Nannarelli, A.: Cached-code compression for energy minimization in embedded processors. In: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (ISLPED), August 2001, pp. 322–327. ACM Press, New York (2001)
14. Brorsson, M., Collin, M.: Adaptive and flexible dictionary code compression for embedded applications. In: Proceedings of the international conference on compilers, architectures and synthesis for embedded systems (CASES), pp. 113–124. ACM Press, New York (2006)
15. Levin, R., Newman, I., Haber, G.: Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In: Stenström, P., Dubois, M., Katevenis, M., Gupta, R., Ungerer, T. (eds.) HiPEAC 2007. LNCS, vol. 4917, pp. 291–304. Springer, Heidelberg (2008)
16. Yang, J., Gupta, R., Zhang, C.: Frequent value encoding for low power data buses. ACM Transactions on Design Automation of Electronic Systems 9(3), 354–384 (2004)
17. Balakrishnan, S., Sohi, G.S.: Exploiting value locality in physical register files. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), December 2003, pp. 265–276. IEEE, Los Alamitos (2003)
18. Brooks, D., Martonosi, M.: Dynamically exploiting narrow width operands to improve processor power and performance. In: Proceedings of the Fifth International Symposium on High-Performance Computer Architecture (HPCA), January 1999, pp. 13–22. IEEE, Los Alamitos (1999)
19. Canal, R., González, A., Smith, J.E.: Software-controlled operand-gating. In: Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO), March 2004, pp. 125–136. IEEE, Los Alamitos (2004)
20. Thuresson, M., Spracklen, L., Stenstrom, P.: Memory-link compression schemes: A value locality perspective. IEEE Transactions on Computers 57(7), 916–927 (2008)

MLP-Aware Runahead Threads in a Simultaneous Multithreading Processor

Kenzo Van Craeynest, Stijn Eyerman, and Lieven Eeckhout

Department of Electronics and Information Systems (ELIS), Ghent University, Belgium
{kevcraey,seyerman,leeckhou}@elis.UGent.be

Abstract. Threads experiencing long-latency loads on a simultaneous multithreading (SMT) processor may clog shared processor resources without making forward progress, thereby starving other threads and reducing overall system throughput. An elegant solution to the long-latency load problem in SMT processors is to employ runahead execution. Runahead threads do not block commit on a long-latency load but instead execute subsequent instructions in a speculative execution mode to expose memory-level parallelism (MLP) through prefetching. The key benefit of runahead SMT threads is twofold: (i) runahead threads do not clog resources on a long-latency load, and (ii) runahead threads exploit far-distance MLP.

This paper proposes MLP-aware runahead threads: runahead execution is only initiated in case there is far-distance MLP to be exploited. By doing so, useless runahead executions are eliminated, thereby reducing the number of speculatively executed instructions (and thus energy consumption) while preserving the performance of the runahead thread and potentially improving the performance of the co-executing thread(s). Our experimental results show that MLP-aware runahead threads reduce the number of speculatively executed instructions by 13.9% and 10.1% for two-program and four-program workloads, respectively, compared to MLP-agnostic runahead threads while achieving comparable system throughput and job turnaround time.

1 Introduction

Long-latency loads (last D-cache level misses and D-TLB misses) have a big performance impact on simultaneous multithreading (SMT) processors [23]. In particular, in an SMT processor with dynamically shared resources, a thread experiencing a long-latency load will eventually stall while holding resources (reorder buffer entries, issue queue slots, rename registers, etc.), thereby potentially starving the other thread(s) and reducing overall system throughput.

Tullsen and Brown [21] recognized this problem and proposed to limit the amount of resources allocated by threads that are stalled due to long-latency loads. In their *flush* policy, fetch is stalled as soon as a long-latency load is detected and instructions are flushed from the pipeline in order to free resources allocated by the long-latency thread. The flush policy by Tullsen and Brown, however, does not preserve memory-level parallelism (MLP) [3,8], but instead serializes independent long-latency loads. This may hurt the performance of memory-intensive (or, more precisely, MLP-intensive) threads. Eyerman and Eeckhout [6] therefore proposed the *MLP-aware flush* policy which first predicts

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 110–124, 2009.

the MLP distance for a long-latency load, i.e., it predicts the number of instructions one needs to go down the dynamic instruction stream for exposing the available MLP. Subsequently, based on the predicted MLP distance, MLP-aware flush decides to (i) flush the thread in case there is no MLP, or (ii) continue allocating resources for the long-latency thread for as many instructions as predicted by the MLP predictor. The key idea is to flush a thread only in case there is no MLP; in case there is MLP, MLP-aware flush allocates as many resources as required to expose the available memory-level parallelism.

Ramirez et al. [17] proposed runahead threads in an SMT processor which avoid resource clogging on long-latency loads while exposing memory-level parallelism. The idea of runahead execution [14] is to not block commit on a long-latency load, but to speculatively execute instructions ahead in order to expose MLP through prefetching. Runahead threads are particularly interesting in the context of an SMT processor because they solve two issues: (i) they do not clog resources on long-latency loads, and (ii) they preserve MLP, and even allow for exploiting far-distance MLP (beyond the scope of the reorder buffer).

A limitation of runahead threads in an SMT processor though is that they consume execution resources (functional unit slots, issue queue slots, reorder buffer entries, etc.) even if there is no MLP to be exploited, i.e., runahead execution does not contribute to the performance of the runahead thread in case there is no MLP to be exploited, and in addition, may hurt the performance of the co-executing thread(s) and thus overall system performance. In this paper, we propose *MLP-aware runahead threads*. The key idea of MLP-aware runahead threads is to enter runahead execution only in case there is far-distance MLP to be exploited. In particular, the MLP distance predictor first predicts the MLP distance upon a long-latency load, and in case the MLP distance is large, runahead execution is initiated. If not, i.e., in case the MLP distance is small, we fetch stall the thread after having fetched as many instructions as predicted by the MLP distance predictor, or we (partially) flush the long-latency thread if more instructions have been fetched than predicted by the MLP distance predictor.

MLP-aware runahead threads reduce the number of speculatively executed instructions significantly over MLP-agnostic runahead threads while not affecting overall SMT performance. Our experimental results using the SPEC CPU2000 benchmarks on a 4-wide superscalar SMT processor configuration report that MLP-aware runahead threads reduce the number of speculatively executed instructions by 13.9% and 10.1% on average for two-program and four-program workloads, respectively, compared to MLP-agnostic runahead threads, while yielding comparable system throughput and job turnaround time. Binary MLP prediction (using the previously proposed MLP predictor by Mutlu et al. [13]) along with an MLP-agnostic flush policy, further reduces the number of speculatively executed instructions under runahead execution by 13% but hurts system throughput (STP) by 11% and job turnaround time (ANTT) by 2.3% on average.

This paper is organized as follows. We first revisit the MLP-aware flush policy (Section 2) and runahead SMT threads (Section 3). Subsequently, we propose MLP-aware runahead threads in Section 4. After detailing our experimental setup in Section 5, we then present our evaluation in Section 6. Finally, we describe related work (Section 7), and conclude (Section 8).

2 MLP-Aware Flush

The MLP-aware flush policy proposed in [6] consists of three mechanisms: (i) it identifies long-latency loads, (ii) it predicts the load's MLP distance, and (iii) it stalls fetch or flushes the long-latency thread based on the predicted MLP distance. The first step is trivial (i.e., a load instruction is labeled as a long-latency load as soon as the load is found out to be an off-chip memory access, e.g., an L3 miss or a D-TLB miss). We now discuss the second and third steps in more detail.

2.1 MLP Distance Prediction

Once a long-latency load is identified, the MLP distance predictor predicts the *MLP distance*, or the number of instructions one needs to go down the dynamic instruction stream in order to expose the maximum exploitable MLP for the given reorder buffer size. The MLP distance predictor consists of a table indexed by the load PC, and each entry in the table records the MLP distance for the corresponding load. There is one MLP distance predictor per thread.

Updating the MLP distance predictor is done using a structure called the *long-latency shift register* (LLSR), see Figure 1. The LLSR has as many entries as there are reorder buffer entries divided by the number of threads (assuming a shared reorder buffer), and there are as many LLSRs as there are threads. Upon committing an instruction from the reorder buffer, the LLSR is shifted over one bit position from tail to head, and one bit is inserted at the tail of the LLSR. A '1' is inserted in case the committed instruction is a long-latency load, and a '0' is inserted otherwise. Along with inserting a '0' or a '1' we also keep track of the load PCs in the LLSR. In case a '1' reaches the head of the LLSR, we update the MLP distance predictor table. This is done by computing the MLP distance which is the bit position of the last appearing '1' in the LLSR when reading the LLSR from head to tail. In the example given in Figure 1, the MLP distance equals 6. The MLP distance predictor is updated by inserting the computed MLP distance in the predictor table entry pointed to by the long-latency load PC. In other words, the MLP distance predictor is a simple last value predictor, i.e., the most recently observed MLP distance is stored in the predictor table.

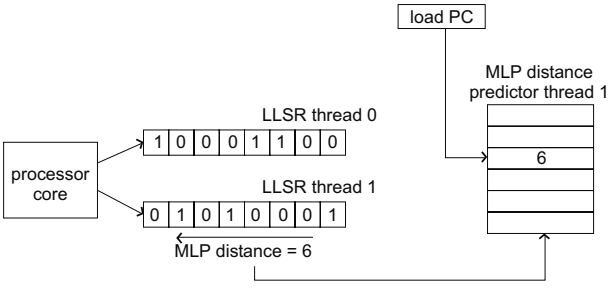


Fig. 1. Updating the MLP distance predictor

2.2 MLP-Aware Fetch Policy

The best performing MLP-aware fetch policy reported in [6] is the MLP-aware flush policy and operates as follows. Say the predicted MLP distance equals *m*. Then, if more than *m* instructions have been fetched since the long-latency load, say *n* instructions, we flush the last *n* - *m* instructions fetched. If less than *m* instructions have been fetched since the long-latency load, we continue fetching instructions until *m* instructions have been fetched, and we then fetch stall the thread.

The flush mechanism requires checkpointing support by the microarchitecture. Commercial processors such as the Alpha 21264 [11] effectively support checkpointing at all instructions. If the microprocessor would only support checkpointing at branches for example, the flush mechanism could flush the instructions past the first branch after the next *m* instructions. The MLP-aware flush policy resorts to the ICOUNT fetch policy [22] in the absence of long-latency loads. The MLP-aware flush policy also implements the 'continue the oldest thread' (COT) mechanism proposed by Cazorla et al. [1]. COT means that in case all threads stall because of a long-latency load, the thread that stalled first gets priority for allocating resources. The idea is that the thread that stalled first is likely to be the first thread to get the data back from memory and continue execution.

3 Runahead Threads

Runahead execution [4,14] avoids the processor from stalling when a long-latency load hits the head of the reorder buffer. When a long-latency load that is still being serviced, reaches the reorder buffer head, the processor takes a checkpoint (which includes the architectural register state, the branch history register and the return address stack), records the program counter of the blocking long-latency load, and initiates runahead execution. The processor then continues to execute instructions in a speculative way past the long-latency load: these instructions do not change the architectural state. Long-latency loads executed during runahead send their requests to main memory but their results are identified as invalid; and an instruction that uses an invalid argument also produces an invalid result. Some of the instructions executed during runahead execution (those that are independent of the long-latency loads) may miss in the cache as well. Their latencies then overlap with the long-latency load that initiated runahead execution. And this is where the performance benefit of runahead comes from: it exploits memory-level parallelism (MLP) [3,8], i.e., independent memory accesses are processed in parallel. When, eventually, the initial long-latency load returns from memory, the processor exits runahead execution, flushes the pipeline, restores the checkpoint, and resumes normal execution starting with the load instruction that initiated runahead execution. This normal execution will make faster progress because some of the data has already been prefetched in the caches during runahead execution.

Whereas Mutlu et al. [14] proposed runahead execution for achieving high performance on single-threaded superscalar processors, Ramirez et al. [17] integrate runahead threads in an SMT processor. The reason for doing so is twofold. First, runahead threads seek for exploiting MLP thereby improving per-thread performance. Second, runahead threads do not stall on commit and thus do not clog resources in an SMT processor.

This appealing solution to the shared resource partitioning problem in SMT processors yields substantial SMT performance improvements, especially for memory-intensive workloads according to Ramirez et al. (and we confirm those results in our evaluation). The runahead threads proposal by Ramirez et al., however, initiates runahead execution upon a long-latency load irrespective of whether there is MLP to be exploited. As a result, in case there is no MLP, runahead execution will consume resources without contributing to performance, i.e., the runahead execution is useless because it does not exploit MLP. This is the problem being addressed in this paper and for which we propose MLP-aware threads as described in the next section.

4 MLP-Aware Runahead Threads

An MLP-aware fetch policy as well as runahead threads come with their own benefits and limitations. The limitation of an MLP-aware fetch policy is that it cannot exploit MLP over large distances, i.e., the exploitable MLP is limited to (a fraction of) the reorder buffer size. Runahead threads on the other hand can exploit MLP at large distances, beyond the scope of the reorder buffer, which improves performance substantially for memory-intensive workloads. However, if MLP-agnostic — as in the original description of runahead execution by Mutlu et al. [14] as well as in the follow-on work by Ramirez et al. [17] — runahead execution is initiated upon every in-service long-latency load that hits the reorder buffer head irrespective of whether there is MLP to be exploited. As a result, runahead threads may consume execution resources without any performance benefit for the runahead thread. Moreover, runahead execution may even hurt the performance of the co-executing thread(s). Another disadvantage of runahead execution compared to the MLP-aware flush policy is that more instructions need to be re-fetched and re-executed upon the return of the initiating long-latency load. In the MLP-aware flush policy on the other hand, instructions reside in the reorder buffer and issue queues and need not be re-fetched, and, in addition, the instructions that are independent of the blocking long-latency load need not be re-executed, potentially saving execution resources and energy consumption.

To combine the best of both worlds, we propose *MLP-aware runahead threads* in this paper. We distinguish two approaches to MLP-aware runahead threads.

Runahead threads based on binary MLP prediction. The first approach is to employ binary MLP prediction. We therefore use the MLP predictor proposed by Mutlu et al. [13] which was originally developed for limiting the number of useless runahead periods, thereby reducing the number of speculatively executed instructions under runahead execution in order to save energy. The idea of employing the MLP predictor is to enter runahead mode only in case the MLP predictor predicts there is far-distance MLP to be exploited.

The MLP predictor by Mutlu et al. is a load-PC indexed table with a two-bit saturating counter per table entry. Runahead mode is entered only in case the counter is in the ‘10’ or ‘11’ states. A long-latency load which has no counter associated with it, allocates a counter and resets the counter (to the state ‘00’). Runahead execution is not entered in the ‘00’ and ‘01’ states; instead, the counter is incremented. During

runahead execution, the processor keeps track of the number of long-latency loads generated. (Mutlu et al. count the number of loads generated beyond the reorder buffer; in the SMT context with a shared reorder buffer, this translates to the reorder buffer size divided by the number of hardware threads.) When exiting runahead mode, if at least one long-latency load was generated during runahead mode, the associated counter is incremented; if not, the counter is decremented if in the '11' state, and is reset if in the '10' state.

Runahead threads based on MLP distance prediction. The second approach to MLP-aware runahead threads is to predict the MLP distance rather than to rely on a binary MLP prediction. We first predict the MLP distance upon a long-latency load. In case the predicted MLP distance is smaller than half the reorder buffer size for a two-thread SMT processor and one fourth the reorder buffer size for a four-thread SMT processor (i.e., this is what the MLP-aware flush policy can exploit), we apply the MLP-aware flush policy. In case the predicted MLP distance is larger than half (or one fourth) the reorder buffer size, we enter runahead mode. In other words, if there is no MLP or if there is exploitable MLP over a short distance only, we reside to the MLP-aware flush policy; if there is large-distance MLP to be exploited, we initiate runahead execution.

5 Experimental Setup

5.1 Benchmarks and Simulator

We use the SPEC CPU2000 benchmarks in this paper with their reference inputs. These benchmarks are compiled for the Alpha ISA using the Compaq C compiler (`cc`) version V6.3-025 with the `-O4` optimization option. For all of these benchmarks we select 200M instruction (early) simulation points using the SimPoint tool [15,18]. We use a wide variety of randomly selected two-thread and four-thread workloads. The two-thread and four-thread workloads are classified as ILP-intensive, MLP-intensive or mixed ILP/MLP-intensive workloads.

We use the SMTSIM simulator v1.0 [20] in all of our experiments. The processor model being simulated is the 4-wide superscalar out-of-order SMT processor shown in Table 1. The default fetch policy is ICOUNT 2.4 [22] which allows up to four instructions from up to two threads to be fetched per cycle. We added a write buffer to the simulator's processor model: store operations leave the reorder buffer upon commit and wait in the write buffer for writing to the memory subsystem; commit blocks in case the write buffer is full and we want to commit a store.

5.2 Performance Metrics

We use two system-level performance metrics in our evaluation: system throughput (STP) and average normalized turnaround time (ANTT) [7]. System throughput (STP) is a system-oriented metric which measures the number of jobs completed per unit of time, and is defined as:

$$STP = \sum_{i=1}^{n} \frac{CPI_i^{ST}}{CPI_i^{MT}},$$

Table 1. The baseline SMT processor configuration

parameter	value
fetch policy	ICOUNT 2.4
pipeline depth	14 stages
(shared) reorder buffer size	128 entries
(shared) load/store queue	64 entries
instruction queues	64 entries in both IQ and FQ
rename registers	100 integer and 100 floating-point
processor width	4 instructions per cycle
functional units	4 int ALUs, 2 ld/st units and 2 FP units
branch misprediction penalty	11 cycles
branch predictor	2K-entry gshare
branch target buffer	256 entries, 4-way set associative
write buffer	8 entries
L1 instruction cache	64KB, 4-way, 64-byte lines
L1 data cache	64KB, 4-way, 64-byte lines
unified L2 cache	512KB, 8-way, 64-byte lines
unified L3 cache	4MB, 16-way, 64-byte lines
instruction/data TLB	128/512 entries, fully-assoc, 8KB pages
cache hierarchy latencies	L2 (11), L3 (35), MEM (500)

with CPI_i^{ST} and CPI_i^{MT} the cycles per instruction achieved for program i during single-threaded and multi-threaded execution, respectively; there are n threads running simultaneously. STP is a higher-is-better metric and equals the weighted speedup metric proposed by Snavely and Tullsen [19].

Average normalized turnaround time (ANNT) is a user-oriented metric which quantifies the average user-perceived slowdown due to multithreading. ANTT is computed as

$$ANTT = \frac{1}{n}\sum_{i=1}^{n}\frac{CPI_i^{MT}}{CPI_i^{ST}}.$$

ANTT equals the reciprocal of the hmean metric proposed in [12], and is a lower-is-better metric. Eyerman and Eeckhout [7] argue that both STP and ANTT should be reported in order to gain insight into how a given multithreaded architecture affects system-perceived and user-perceived performance, respectively.

When simulating a multi-program workload, simulation stops when 400 million instructions have been executed. At that point, program i will have executed x_i million instructions. The single-threaded CPI_i^{ST} used in the above formulas equals single-threaded CPI after x_i million instructions. When we report average STP and ANTT numbers across a number of multi-program workloads, we use the harmonic and arithmetic mean for computing the average STP and ANTT, respectively, following the recommendations on the use of averages by John [10].

5.3 Hardware Cost

The performance numbers reported in the evaluation section assume the following hardware costs. For both the binary MLP predictor and the MLP distance predictor we

assume a PC-indexed 2K-entry table. (We experimented with a number of predictor configurations, including the tagged set-associative table organization proposed by Mutlu et al. [13] and we found the untagged 2K-entry to slightly outperform the tagged organization by Mutlu et al.) An entry in the binary MLP predictor is a 2-bit field following Mutlu et al. [13]. An entry in the MLP distance predictor is a 3-bit field; one bit encodes whether long-distance MLP is to be predicted, and the other two bits encode the MLP distance within the reorder buffer in buckets of 16 instructions. The hardware cost for a run-length encoded LLSR equals 0.7Kbits in total: 32 (maximum number of outstanding long-latency loads) times 22 bits (11 bits for keeping track of the load PC index in the 2K-entry MLP distance predictor, plus 11 bits for the encoded run length — maximum of 2048 instructions — since the prior long-latency load miss). In summary, the total hardware cost for the binary MLP predictor equals 4Kbits; the total hardware cost for the MLP distance predictor (predictor table plus LLSR) equals 6.7Kbits.

6 Evaluation

6.1 MLP Distance Predictor

Key to the success of MLP-aware runahead threads is the accuracy of the MLP distance predictor. The primary concern is whether the predictor can accurately estimate far-distance MLP in order to decide whether or not to go in runahead mode.

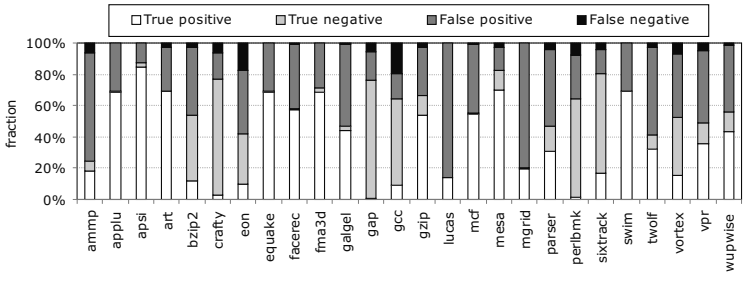


Fig. 2. Quantifying the accuracy of the MLP distance predictor

Figure 2 shows the accuracy of the MLP distance predictor. A true positive denotes correctly predicted long-distance MLP and a true negative denotes correctly predicted short-distance or no MLP; the false positives and false negatives denote mispredictions. The prediction accuracy equals 61% on average, and the majority of mispredictions are false positives. In spite of this relatively low prediction accuracy, MLP-aware runahead threads are effective as will be demonstrated in the next few paragraphs. Improving MLP distance prediction will likely lead to improved effectiveness of MLP-aware runahead threads, i.e., reducing the number of false positives will reduce the number of speculatively executed instructions and will thus increase energy saving opportunities — this is left for future work though.

6.2 Two-Program Workloads

We compare the following SMT fetch policies and architectures:

- ICOUNT [22] which strives at having an equal number of instructions from all threads in the front-end pipeline and instruction queues. The following fetch policies extend upon the ICOUNT policy.
- The *MLP-aware flush* approach [6] predicts the MLP distance m for a long-latency load, and fetch stalls or flushes the thread after m instructions since the long-latency load.
- *Runahead threads*: threads go in runahead mode when the oldest instruction in the reorder buffer is a long-latency load that is still being serviced [17].
- *Binary MLP-aware runahead threads w/ ICOUNT*: the binary MLP predictor by Mutlu et al. [13] predicts whether there is far-distance MLP to be exploited, and a thread only goes in runahead mode in case MLP is predicted. In case there is no (predicted) MLP, we resort to ICOUNT.
- *Binary MLP-aware runahead threads w/ flush*: this is the same policy as the one above, except that in case of no (predicted) MLP, we perform a flush. The trade-off between this policy and the latter is that ICOUNT may exploit short-distance MLP whereas flush does not, however, flush prevents resource clogging.
- *MLP-distance-aware runahead threads*: the MLP distance predictor by Eyerman and Eeckhout [6] predicts the MLP distance. If there is far-distance MLP to be exploited, the thread goes in runahead mode. If there is only short-distance MLP to be exploited, the thread is fetch stalled and/or flushed according to the predicted MLP distance.

Figures 3 and 4 compare these six fetch policies in terms of the STP and ANTT performance metrics, respectively, for the two-program workloads. These results confirm the results presented in prior work by Ramirez et al. [17]: runahead threads improve both system throughput and job turnaround time significantly over both ICOUNT and MLP-aware flush: STP and ANTT improve by 70.1% and 43.8%, respectively, compared to ICOUNT; and STP and ANTT improve by 44.3% and 26.8%, respectively, compared to MLP-aware flush. These results also show that MLP-aware runahead threads (rightmost bars) achieve comparable performance as MLP-agnostic runahead threads. Moreover, MLP-aware runahead threads achieve a slight improvement in both STP and ANTT for some workloads over MLP-agnostic runahead threads, e.g., mesa-galgel achieves a 3.3% higher STP and a 3.2% smaller ANTT under MLP-aware runahead threads compared to MLP-agnostic runahead threads. The reason for this performance improvement is that preventing one thread from entering runahead mode gives more resources to the co-executing thread thereby improving the performance of the co-executing thread. For other workloads, on the other hand, MLP-aware runahead threads result in slightly worse performance compared to MLP-agnostic runahead threads, e.g., the worst performance is observed for art-mgrid: 3% reduction in STP and 0.3% increase in ANTT. These performance degradations are due to incorrect MLP distance predictions.

Figures 3 and 4 also clearly illustrate the effectiveness of MLP distance prediction versus binary MLP prediction. The MLP distance predictor is more effective than the

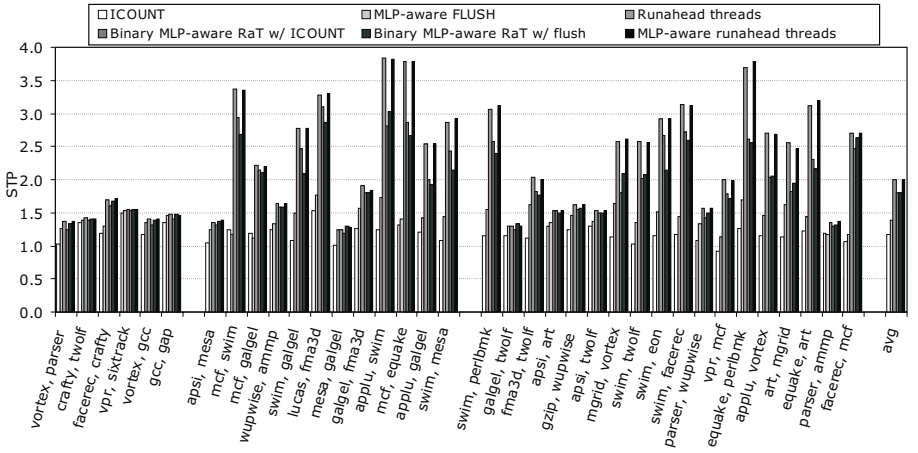


Fig. 3. Comparing MLP-aware runahead threads against other fetch SMT policies in terms of STP for two-program workloads: ILP-intensive workloads are shown on the left, MLP-intensive workloads are shown in the middle and mixed ILP/MLP-intensive workloads are shown on the right

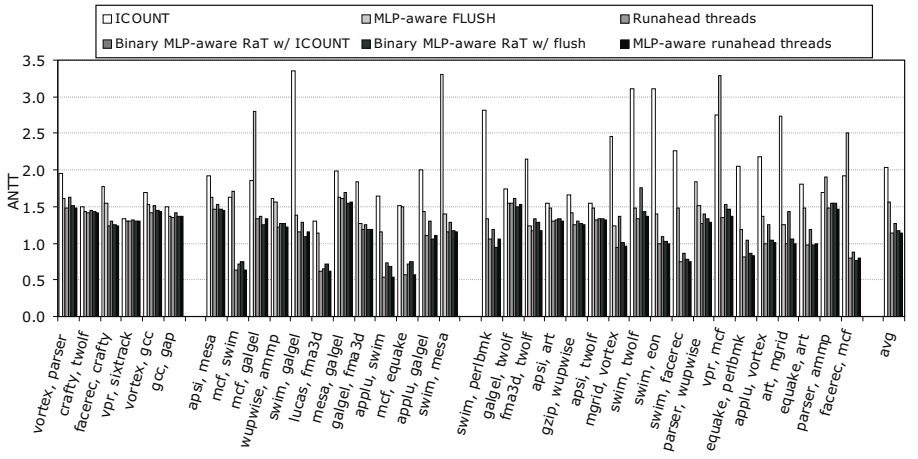


Fig. 4. Comparing MLP-aware runahead threads against other fetch SMT policies in terms of ANTT for two-program workloads: ILP-intensive workloads are shown on the left, MLP-intensive workloads are shown in the middle and mixed ILP/MLP-intensive workloads are shown on the right

binary MLP predictor proposed by Mutlu et al. [13]: i.e., STP improves by 11% on average and ANTT improves by 2.3% compared to the binary MLP-aware policy with flush; compared to the binary MLP-aware policy with ICOUNT, the MLP distance predictor improves STP by 11.5% and ANTT by 10%. The reason is twofold. First, the LLSR employed by the MLP distance predictor continuously monitors the MLP distance for each long-latency load. The binary MLP predictor by Mutlu et al. only checks for far-distance MLP through runahead execution; as runahead execution is not initiated

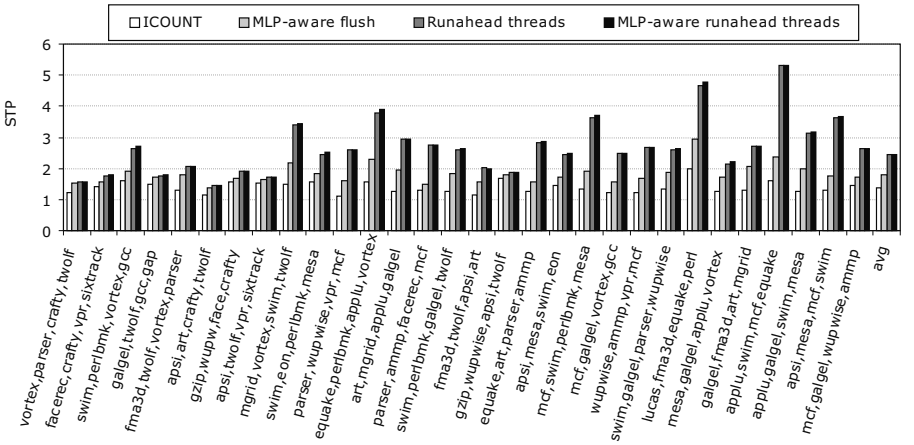


Fig. 5. Comparing MLP-aware runahead threads against other fetch SMT policies in terms of STP for four-program workloads

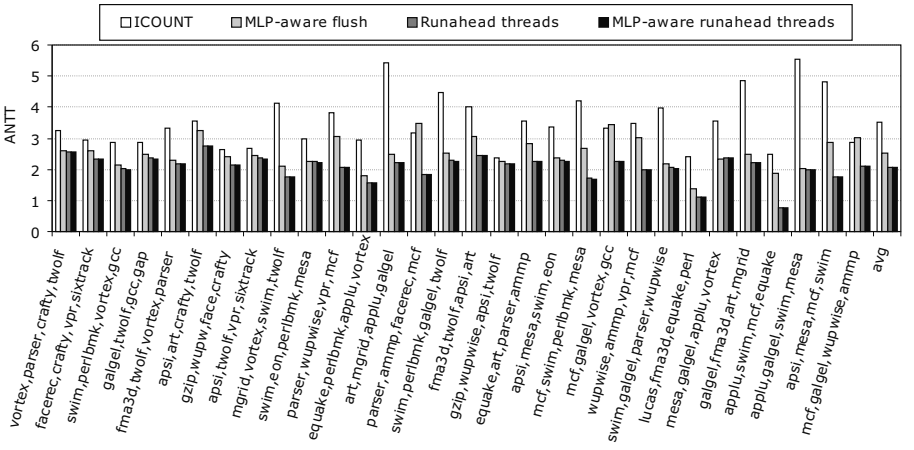


Fig. 6. Comparing MLP-aware runahead threads against other fetch SMT policies in terms of ANTT for four-program workloads

for each long-latency load, it provides partial MLP information only. Second, the MLP distance predictor releases resources allocated by the long-latency thread as soon as the short-distance MLP (within half the reorder buffer) has been exploited. The binary MLP-aware policy on the other hand clogs resources (through the ICOUNT mechanism) or does not exploit short-distance MLP (through the flush policy).

6.3 Four-Program Workloads

Figures 5 and 6 show STP and ANTT, respectively, for the four-program workloads. The overall conclusion is similar as for two-program workloads: MLP-aware runahead threads achieve similar performance as MLP-agnostic runahead threads. The

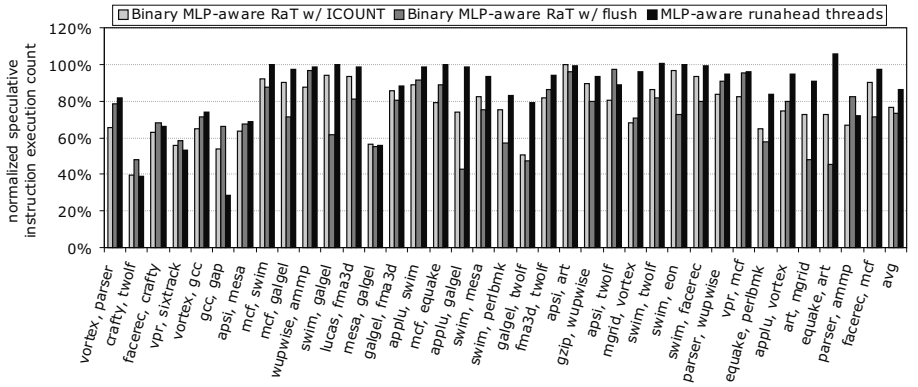


Fig. 7. Normalized speculative instruction count compared to MLP-agnostic runahead threads for the two-program workloads

performance improvements are slightly higher though for the four-program workloads than for the two-program workloads because the co-executing programs compete more for the shared resources on a four-threaded SMT processor than on a two-threaded SMT processor. Making the runahead threads MLP-aware provides more shared resources for the co-executing programs which improves both single-program performance as well as overall system performance.

6.4 Reduction in Speculatively Executed Instructions

As mentioned before, the main motivation for making runahead MLP-aware is to reduce the number of useless runahead executions, and thereby reduce the number of speculatively executed instructions under runahead execution in order to reduce energy consumption. Figure 7 quantifies the normalized number of speculatively executed instructions compared to MLP-agnostic runahead threads. MLP-aware runahead threads reduce the number of speculatively executed instructions by 13.9% on average; this is due to eliminating useless runahead execution periods. (We obtain similar results for the four-program workloads with an average 10.1% reduction in the number of speculatively executed instructions; these results are not shown here because of space constraints.) Binary MLP-aware runahead threads with ICOUNT and flush achieve higher reductions in the number of speculatively executed instructions (23.7% and 27%, respectively), however, this comes at the cost of reduced performance (by 11% to 11.5% in STP and 2.3% to 10% in ANTT) as previously shown.

7 Related Work

There are two ways of partitioning the resources in an SMT processor. One approach is static partitioning [16] as done in the Intel Pentium 4 [9], in which each thread gets an equal share of the resources. Static partitioning solves the long-latency load problem: a long-latency thread cannot clog resources, however, it does not provide flexibility: a resource that is not being used by one thread cannot be used by the other thread(s).

The second approach, called dynamic partitioning, on the other hand provides flexibility by allowing multiple threads to share resources, however, preventing long-latency threads from clogging resources is a challenge. In dynamic partitioning, the fetch policy typically determines what thread to fetch instructions from in each cycle and by consequence, the fetch policy also implicitly manages the shared resources. Several fetch policies have been proposed in the recent literature. ICOUNT [22] prioritizes threads with fewer instructions in the pipeline. The limitation of ICOUNT is that in case of a long-latency load, ICOUNT may continue allocating resources for the blocking long-latency thread; by consequence, these resources will be hold by the blocking thread and will prevent the other thread(s) from allocating these resources. In response to this problem, Tullsen and Brown [21] proposed two schemes for handling long-latency loads, namely (i) fetch stall the long-latency thread, and (ii) flush instructions fetched passed the long-latency load in order to deallocate resources. Cazorla et al. [1] improved upon the work done by Tullsen and Brown by predicting long-latency loads along with the 'continue the oldest thread (COT)' mechanism that prioritizes the oldest thread in case all threads wait for a long-latency load. Eyerman and Eeckhout [6] made the flush policy MLP-aware in order to preserve the available MLP upon a flush or fetch stall on a long-latency thread.

An alternative approach is to drive the fetch policy through explicit resource partitioning. For example, Cazorla et al. [2] propose DCRA which monitors the dynamic usage of resources by each thread and strives at giving a higher share of the available resources to memory-intensive threads. The input to their scheme consists of various usage counters for the number of instructions in the instruction queues, the number of allocated physical registers and the number of L1 data cache misses. Using these counters, DCRA dynamically determines the amount of resources required by each thread and prevents threads from using more resources than they are entitled to. However, DCRA drives the resource partitioning mechanism using imprecise MLP information and allocates a fixed amount of additional resources for memory-intensive workloads irrespective of the amount of MLP.

El-Moursy and Albonesi [5] propose to give fewer resources to threads that experience many data cache misses in order to minimize issue queue occupancies for saving energy. They propose two schemes for doing so, namely data miss gating (DG) and predictive data miss gating (PDG). DG drives the fetching based on the number of observed L1 data cache misses, i.e., by counting the number of L1 data cache misses in the execute stage of the pipeline. When the number of L1 data cache misses exceeds a given threshold, the thread is fetch gated. PDG strives at overcoming the delay between observing the L1 data cache miss and the actual fetch gating in the DG scheme by predicting L1 data cache misses in the front-end pipeline stages.

8 Conclusion

Runahead threads solve the long-latency load problem in an SMT processor elegantly by exposing (far-distance) memory-level parallelism while not clogging shared processor resources. A limitation though of existing runahead SMT execution proposals is that runahead execution is initiated upon a long-latency load irrespective of whether there is

far-distance MLP to be exploited. A useless runahead execution, i.e., one along which there is no exploitable MLP, thus wastes execution resources and energy.

This paper proposed MLP-aware runahead threads to reduce the number of useless runahead periods. In case the MLP distance predictor predicts there is far-distance MLP to be exploited, the long-latency thread enters runahead execution. If not, the long-latency thread is flushed or fetch stalled per the predicted MLP distance. By doing so, runahead execution consumes resources only in case of long-distance MLP; if not, the MLP-aware flush policy frees allocated resources while exposing short-distance MLP, if available. Our experimental results report an average reduction of 13.9% in the number of speculatively executed instructions compared to MLP-agnostic runahead threads for two-program workloads while incurring no performance degradation; for four-program workloads, we report a 10.1% reduction in the number of speculatively executed instructions. Previously proposed binary MLP prediction achieves greater reductions in the number of speculatively executed instructions (by 23.7% to 27% on average) compared to MLP-agnostic runahead threads, however, it incurs an average 11% to 11.5% reduction in system throughput and an average 2.3% to 10% reduction in average job turnaround time.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. Stijn Eyerman and Lieven Eeckhout are postdoctoral fellows with the Fund for Scientific Research in Flanders (Belgium) (FWO-Vlaanderen). Additional support is provided by the FWO projects G.0160.02 and G.0255.08.

References

1. Cazorla, F.J., Fernandez, E., Ramirez, A., Valero, M.: Optimizing long-latency-load-aware fetch policies for SMT processors. International Journal of High Performance Computing and Networking (IJHPCN) 2(1), 45–54 (2004)
2. Cazorla, F.J., Ramirez, A., Valero, M., Fernandez, E.: Dynamically controlled resource allocation in SMT processors. In: MICRO, pp. 171–182 (December 2004)
3. Chou, Y., Fahs, B., Abraham, S.: Microarchitecture optimizations for exploiting memory-level parallelism. In: ISCA, pp. 76–87 (June 2004)
4. Dundas, J., Mudge, T.: Improving data cache performance by pre-executing instructions under a cache miss. In: ICS, pp. 68–75 (July 1997)
5. El-Moursy, A., Albonesi, D.H.: Front-end policies for improved issue efficiency in SMT processors. In: HPCA, pp. 31–40 (February 2003)
6. Eyerman, S., Eeckhout, L.: A memory-level parallelism aware fetch policy for SMT processors. In: HPCA, pp. 240–249 (February 2007)
7. Eyerman, S., Eeckhout, L.: System-level performance metrics for multi-program workloads. IEEE Micro. 28(3), 42–53 (2008)
8. Glew, A.: MLP yes! ILP no! In: ASPLOS Wild and Crazy Idea Session (October 1998)
9. Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., Roussel, P.: The microarchitecture of the Pentium 4 processor. Intel. Technology Journal Q1 (2001)
10. John, L.K.: Aggregating performance metrics over a benchmark suite. In: John, L.K., Eeckhout, L. (eds.) Performance Evaluation and Benchmarking, pp. 47–58. CRC Press, Boca Raton (2006)

11. Kessler, R.E., McLellan, E.J., Webb, D.A.: The Alpha 21264 microprocessor architecture. In: ICCD, pp. 90–95 (October 1998)
12. Luo, K., Gummaraju, J., Franklin, M.: Balancing throughput and fairness in SMT processors. In: ISPASS, pp. 164–171 (November 2001)
13. Mutlu, O., Kim, H., Patt, Y.N.: Techniques for efficient processing in runahead execution engines. In: ISCA, pp. 370–381 (June 2005)
14. Mutlu, O., Stark, J., Wilkerson, C., Patt, Y.N.: Runahead execution: An alternative to very large instruction windows for out-of-order processors. In: HPCA, pp. 129–140 (February 2003)
15. Perelman, E., Hamerly, G., Calder, B.: Picking statistically valid and early simulation points. In: Malyshkin, V.E. (ed.) PaCT 2003. LNCS, vol. 2763, pp. 244–256. Springer, Heidelberg (2003)
16. Raasch, S.E., Reinhardt, S.K.: The impact of resource partitioning on SMT processors. In: Malyshkin, V.E. (ed.) PaCT 2003. LNCS, vol. 2763, pp. 15–26. Springer, Heidelberg (2003)
17. Ramirez, T., Pajuelo, A., Santana, O.J., Valero, M.: Runahead threads to improve SMT performance. In: HPCA, pp. 149–158 (February 2008)
18. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: ASPLOS, pp. 45–57 (October 2002)
19. Snavely, A., Tullsen, D.M.: Symbiotic jobscheduling for simultaneous multithreading processor. In: ASPLOS, pp. 234–244 (November 2000)
20. Tullsen, D.: Simulation and modeling of a simultaneous multithreading processor. In: Proceedings of the 22nd Annual Computer Measurement Group Conference (December 1996)
21. Tullsen, D.M., Brown, J.A.: Handling long-latency loads in a simultaneous multithreading processor. In: MICRO, pp. 318–327 (December 2001)
22. Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L.: Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In: ISCA, pp. 191–202 (May 1996)
23. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous multithreading: Maximizing on-chip parallelism. In: ISCA, pp. 392–403 (June 1995)

IPC Control for Multiple Real-Time Threads on an In-Order SMT Processor

Jörg Mische, Sascha Uhrig, Florian Kluge, and Theo Ungerer

Members of the EC Network of Excellence HiPEAC
Institute of Computer Science
University of Augsburg
86159 Augsburg, Germany
{mische,uhrig,kluge,ungerer}@informatik.uni-augsburg.de

Abstract. This paper proposes an architecture for concurrent scheduling of hard, soft and non real-time threads in embedded systems. It is based on a superscalar in-order processor binary compatible to the Infineon TriCore. The architecture allows a tight static WCET analysis of hard real-time threads. To provide high performance anyway, the absence of speculative elements like branch prediction and out-of-order execution is compensated by multithreading, transforming the processor into an in-order SMT processor.

The Priority Controller that manages the scheduling is able (1) to assign fixed portions of time to hard real-time threads, (2) to control the IPC of soft real-time threads and (3) to fairly distribute execution cycles to non real-time threads. It is located within a separate unit outside the pipeline to avoid prolonging the critical path.

We evaluate the processor using the EEMBC automotive benchmarks and show that the overlapping of two soft real-time threads can be used to either reduce the clock rate by 23% or to grant each thread 65% of its single-threaded IPC. Even if a hard real-time thread is executed predominantly, the remaining resources can be used by concurrent soft real-time threads which reach a performance of 70% compared to their single-threaded execution.

1 Introduction

Complex systems embedded in airplanes, cars and other industrial machinery contain dozens of small microcontrollers, each one different and specially designed for a certain purpose. To reduce costs, a current trend is to reduce the number of microcontrollers by applying fewer microcontrollers of higher performance, e. g. the so-called *domain-based architecture* proposed by Siemens VDO [1].

Therefore future embedded microprocessors must execute multiple threads in parallel. The most crucial challenge comes from the variety of threads: short or long runtime; sporadic or periodic release; hard, soft or non real-time demands.

We present an architecture that is able to schedule this kind of mixed real-time application on an SMT processor in order to maximize throughput while meeting all deadlines. The contributions of this paper are:

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 125–139, 2009.

- A soft real-time scheduler that directly controls the IPC of multiple threads. It is able to run a single thread with up to 100% of its single-threaded IPC, while using spare resources by concurrent threads.
- A compatible round robin policy that allows fair scheduling for non real-time threads. It uses the hardware extensions of the soft real-time scheduler and thus needs no additional hardware.
- The soft real-time scheduler can be combined with a hard real-time scheduler (published in [2]) to provide scheduling policies for mixed real-time demands.

The rest of the paper is organized as follows: in the next section the related work is presented and section 3 gives an outline of the baseline processor and the thread model. Section 4 discusses the limitations of the baseline architecture, while section 5 gives a solution for soft real-time scheduling. The results of the evaluation follow in section 6 and section 7 concludes the paper.

2 Related Work

Early SMT implementations focus on a high overall throughput, not on a predictable distribution of execution time [3,4]. As these SMT processors are enhancements of out-of-order superscalar processors, they are out-of-order architectures, too. The scheduling decision takes place in the fetch stage and is driven by information from the back-end of the pipeline [3].

Raasch et al. [4] improve the performance of a foreground thread by prioritizing it over the background threads. Real-time scheduling is not possible because of the out-of-order pipeline that did not completely eliminate thread interference. They also use time-slicing of the highest priority, but only to increase fairness, not for concurrent real-time threads. The first paper directly addressing soft real-time for SMT processors [5] discusses only co-scheduling issues like the distribution of workloads to thread slots.

Later proposals use explicit resource allocation to reach a target execution time of a foreground thread, while using the spare resources for background threads [6,7,8]. Dorai and Yeung [9] try to preserve, as much as possible, the single-thread performance of the foreground thread, while allowing some progress to the background threads. They reach 97% foreground performance and 50% background performance compared to single-threaded execution.

The scheduler of Cazorla et al. [6] dynamically varies the amount of resources given to the foreground thread to reach a given target IPC. If the target IPC is between 30% and 80% of the single-threaded IPC the deviation is less than 1%. Exactly the same results ($\pm 1\%$ if target IPC $< 80\%$) are published by Yamasaki et al. [7], who control the IPC by monitoring the resource usage and adjusting the fetch priorities. Our scheduler is superior to these IPC controllers, as it *directly* controls the IPC and therefore *exactly* reaches the target IPC (as far as the processor is able to reach this IPC in single-threaded mode).

The *Komodo* processor [10] uses the technique of dilating hard real-time threads and reduces the scheduling to a short round of 100 cycles. Brinkschulte et al. [11] developed a closed control loop to control the IPC of a thread within

the *Komodo*. While Komodo is only a scalar multithreaded processor, the *Real-time Virtual Multiprocessor (RVMP)* [12] uses dilating to schedule hard real-time threads on an SMT processor. But as its schedule is statically precalculated, it cannot use dynamically occurring latencies.

The *Virtual Simple Architecture (VISA)* [13] guarantees the execution time of a simple hypothetical processor, but executes the threads on a high-performance, speculative processor. The progress of the speculative execution is monitored and compared to the guaranteed performance. If it is lower, the processor falls back to a direct, cycle-accurate emulation of the simple architecture to meet the deadlines. If the deadline is of wide scope, VISA can be used to further increase the scope. But if there is only a small scope, there is not enough time to try the high-performance execution, because it must switch to the simple mode as soon as the remaining scope meets the WCET of the simple architecture. By contrast, our approach improves the real-time performance even with tight timing bounds.

3 Background

3.1 Baseline Architecture

The baseline architecture for our research is CarCore [2], an in-order SMT processor. It extends the superscalar in-order processor Infineon TriCore [14] by the ability to schedule more than one thread simultaneously in one cycle. Responsible for assigning multiple threads to multiple functional units is the *Real-time Issue Stage*, located between the fetch and the decode stage (Fig.1). It is driven by a simple priority scheduling policy to allow a fast scheduling decision within one cycle even at high clock rates.

The priorities of the thread slots are not fixed, but can be altered at every clock cycle, strictly speaking an additional unit, the so-called *Priority Controller*,

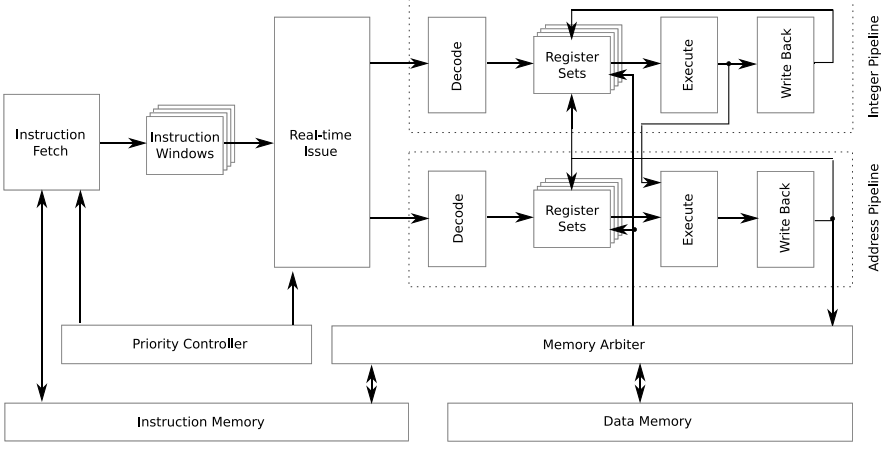


Fig. 1. CarCore Pipelines

provides the priorities and recalculates them in every cycle. The separation of issuing and priority calculation allows complex, time-consuming scheduling algorithms to be implemented within an SMT pipeline.

3.2 Periodic Task Model

A real-time system is typically described by the periodic task model [15], where a system consists of multiple periodic threads, which form a task set. Each thread is characterized by two parameters: the time between the release of two consecutive instances of the same thread (*period*) and the time by which an instance has to complete (*deadline*). For tractability the deadline is generally set equal to the period, i. e. a thread has to finish before its next instance is released, but each thread T_i can have an individual period p_i.

If all threads of a task set meet their deadlines, it is called a schedulable *hard* real-time system. If the deadlines are usually met, but sporadic deadline misses cannot be excluded (and do not harm), it is a *soft* real-time system. To determine, if a task set is schedulable a third thread parameter is important, the upper bound of the thread's execution times, the *Worst Case Execution Time (WCET)* w_i.

Classic hard real-time scheduling policies like earliest deadline first (EDF) [15] examine the task set at every thread suspend or release and switch thread execution if necessary. Therefore the frequency and duration of context switches is unsteady, especially for unequal periods.

3.3 Dilated Threads

We use another method to distribute the execution time, also used by the *Komodo* processor [10] and the *RVMP* [12]: The execution time is divided into small intervals of time, called a round. The length of a round R is given by the greatest common divisor of the periods

$$R = gcd(p_1, ..., p_N) \tag{1}$$

Thereby the period of each thread can be expressed in number of rounds

$$p_i = n_i \cdot R \quad , n_i \in \mathbb{N} \tag{2}$$

In a second step, the WCET of each thread is broken down to each round, resulting in a *Cycle Quantum* c_i,

$$c_i = \frac{w_i}{n_i} = \frac{w_i \cdot R}{p_i} \tag{3}$$

If each thread is executed for c_i cycles during a round, it will definitely meet its deadline, Fig.2 illustrates this dilation of a thread. With this method, scheduling decisions are reduced to decisions within one round.

We call this scheduling policy *Dominant Time Sharing (DTS)* described in [2]. The architecture provides one *Dominant Meta Thread (DMT)* that is executed

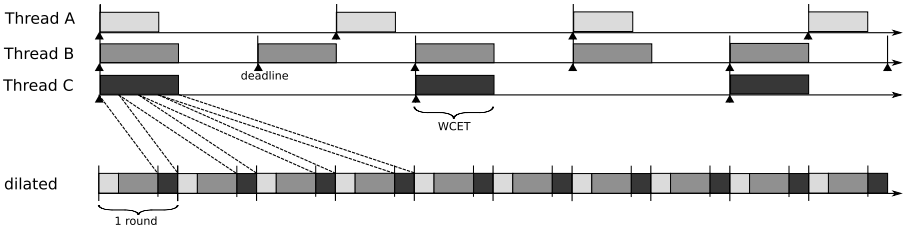


Fig. 2. The periodic task model and dilated threads

as if it were the only thread on the processor. Its runtime behavior is the same as on a single-threaded processor and therefore a static WCET analysis is possible, which in turn allows hard real-time scheduling. Further thread slots are scheduled by fixed priorities and use the remaining processor resources to increase throughput. For multiple hard real-time threads, time-sharing is used to divide the DMT into several hard real-time threads. Within one round, each hard real-time thread is the dominant thread for a fixed number of cycles, its cycle quantum c_i. The ratio $\frac{c_i}{R}$ matches the *utilization*, i. e. the fraction of execution time a thread in the periodic task model demands.

For hard real-time systems, a WCET analysis is inalienable, therefore a certain period of time (measured in cycles) must be guaranteed per round. But depending on the application, control of the really executed instructions (measured by instructions per cycle, IPC) suits the needs of soft real-time threads better [6,7]. Integrated into our model, a requested IPC of b_i results in a requested number of executed instructions per round q_i, called *Instruction Quantum*

$$q_i = b_i \cdot R \tag{4}$$

The underlying performance measurement (cycles or IPC) can be set individually per thread, providing maximum flexibility to minimize the gap between real execution time and deadline, resulting in larger schedulable task sets or a lower required clock rate for a given task set. Using the instruction quantum instead of the cycle quantum for soft real-time threads is the proposal of this paper.

4 Parallel Execution of Soft Real-Time Threads

The DTS scheduling algorithm permits multiple *hard* real-time threads, but it uses the potential of SMT architectures only to a small extent. The only advantage over single-threaded architectures is the chance of executing non real-time threads concurrently, but the hard real-time threads themselves do not benefit from the SMT architecture.

Applying the same algorithm to a single-threaded superscalar processor would yield the same performance apart from the duration of context switches. But the context switch times are constant and small compared to the length of a round and could easily be compensated by a minor increase in clock frequency (which would be possible due to a simpler processor layout).

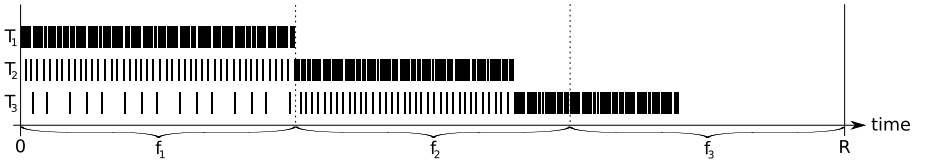


Fig. 3. Three soft real-time threads T_1, T_2, T_3, each requiring one third of the execution time ($c_1 = c_2 = c_3 = \frac{R}{3}$), are overlapped within one round

Parallel execution of threads by resource sharing - the major advantage of multithreaded processors - is not applied on different hard real-time threads, but only on one hard real-time thread and several non real-time threads. Overlapped execution of hard real-time threads is not possible, as the worst case must be considered, i.e. a thread demands all existent resources and leaves no spare resources for concurrent threads. If all hard real-time threads have these high resource claims, only one thread at a time can be executed and therefore time slicing is the only possible solution for concurrent execution of hard real-time threads. Because of this limitation, we focus on *soft* real-time requirements.

4.1 Parallel Execution of Dilated Threads

The method of dividing the processing time into rounds and distributing a certain fraction of it to every real-time thread is preserved. But instead of the disjunctive execution of real-time threads all real-time thread slots are activated at the beginning of a round.

Apparently one thread has the highest priority and thus is executed predominantly. Only if latencies arise during its execution (because of memory accesses or branches), the thread with the second highest priority can use the cycle for execution. As an SMT processor has the ability to assign different functional units to different threads within one single cycle, even functional units useless for the priority thread can be occupied by lower priority threads. Consequently, one thread runs at full speed at the beginning of a round and during its latencies the lower priority threads already complete parts of their jobs.

As soon as the highest priority thread reached its fraction of the round, it is suspended and the next real-time thread runs at maximum speed. The now dominant thread already executed some of its instructions and therefore runs for less cycles than given by its initial quantum. As the savings accumulate the last real-time thread finishes its fraction several cycles before the actual end of the round (see Fig. 3).

If the real-time tasks constantly finish prior to the end of the round, either the remaining time can be used for an additional soft real-time thread or the clock rate can be decreased. By lowering the clock rate the number of cycles per round is reduced and the spare cycles can be eliminated. This saves energy while the threads still meet their deadlines.

4.2 Problems of Cycle Counting

The execution time of a thread is measured by a virtual clock that counts the cycles. In a multithreaded system, not all threads are executed at the same speed and therefore each thread needs its own virtual clock that measures its execution time.

If a disjoint time-sharing scheduler (like DTS described in section 3.3) is used, only the virtual clock of the active thread is enabled while the other clocks are disabled. As soon as the active clock reaches its quantum (i. e. the respective virtual clock reaches the end of its time interval), the clock is stopped, the scheduler switches to the next thread and the appropriate virtual clock is resumed.

The timing becomes more difficult, if the threads overlap. Similar to the time-sharing scheduler, a thread is suspended when the granted time on the virtual clock has passed, but counting the virtual time is more challenging. At each cycle there has to be a decision, which clocks to stop and which clocks to advance.

To present the problem we consider different short instruction traces of a two-way SMT architecture (Fig. 4). Each grid shows the occupation of two different functional units (horizontally) in certain cycles (vertically). Thread X has the highest priority, its single-threaded trace is shown in the leftmost grid and it never changes, no matter which other thread is co-scheduled. The single-threaded traces of the co-scheduled threads A-D are shown in the upper row, the co-scheduled traces in the lower row. The numbers next to the traces are the virtual clocks of the low priority thread.

As the total absolute execution time of a thread is not of relevance for the scheduler and the smallest period of time is a cycle, a cycle counter is used to measure the virtual time. At the beginning of a round the counter is initialized to the quantum of the corresponding thread and as virtual time goes by the counter is decreased until it reaches 0 and the thread is suspended.

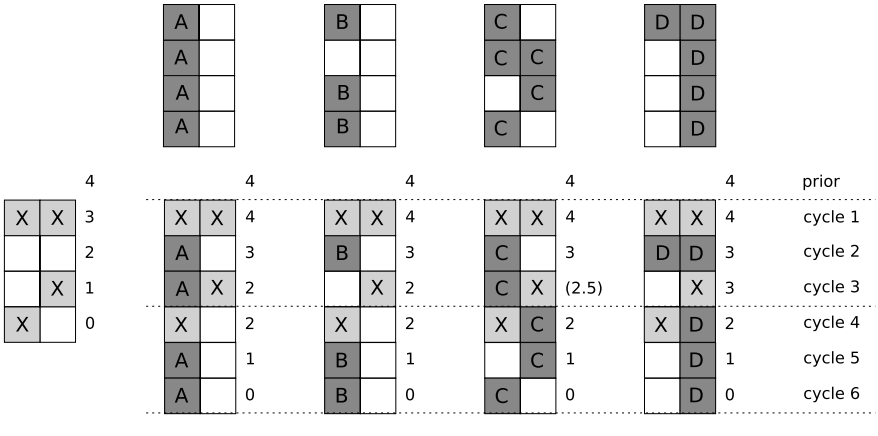


Fig. 4. Overlapping the high priority thread X with different other threads (A-D)

It is easy to count the virtual clock of the thread with the highest priority (X), because it is executed regardless of the other threads, as if it were the only one. Its execution time corresponds exactly to its single-threaded execution time (leftmost grid) and hence its virtual clock corresponds to the real clock. Consequently the virtual clock counter, which equals the quantum (4) at the beginning of the round, is decreased by one in every cycle and reaches 0 after 4 cycles (first column of Fig. 4).

The clock update is more tricky for a co-scheduled thread. As its priority is lower, its execution is delayed in several cycles (thread A, cycles 1 and 4 in Fig. 4). At these cycles, the counter of thread A may not be decremented (its virtual clock stalls). Such a stall cycle appears if a thread with a higher priority occupies a functional unit, that the lower priority thread should use in the same cycle, too. But if the lower priority thread has some latencies (e.g. thread B in cycle 3), its counter must be decremented anyway (i. e. its virtual clock continues), no matter if other threads (of any priority) use the cycle or not.

This relatively simple algorithm (decrement on issue or latency) is sufficient for scalar multithreaded architectures [10,16], but superscalar multithreaded (SMT) architectures pose another problem, as instructions that use different functional units can be executed simultaneously within one cycle.

In cycle 3 the high priority thread X uses only one functional unit, the other one could be used by another thread. But thread C would occupy both functional units, if executed single-threaded (which was assumed when calculating its WCET). Therefore the cycle counter may not be decreased until the whole virtual cycle (using both functional units) was completed. The length of a virtual cycle is not fixed to two, but depends on the surrounding instructions, e.g. in cycle 4 thread X and thread D request only one functional unit each, hence both counters are decreased.

To correctly update the clock it must be determined, how many functional units a thread would occupy in single-threaded mode, i.e. how many instructions can be issued simultaneously. As soon as all of these instructions are executed, the virtual cycle counter can be decreased.

5 Instruction Counting

Besides counting the granted cycles, counting the really executed instructions is another possibility to measure the progress of a thread. But there is no fixed relation between the number of executed instructions and elapsed time. When counting cycles, every cycle corresponds to a fixed time interval, however the duration of an instruction depends on the kind of the instruction and relates to a variable number of cycles and thus a variable period of time.

Therefore it is very difficult to grant a fixed number of executed instructions within a fixed time interval (one round). Strictly speaking, the maximum duration of any instruction must be assumed to all instructions within one round, leading to a very low number of instructions per round. This practice uses only a small fraction of the computing time, as in average the instructions are executed much faster.

Because of the big gap between worst and average case, scheduling by instruction counting is improper for hard real-time threads, time-sharing as presented in section 3.3 is much more applicable. But if it is exceptionally allowed not to reach the instruction quantum within a round, i.e. only demanding soft real-time, the procedure suits well.

5.1 Soft Real-Time

As both overlapping and instruction counting only achieve soft real-time demands, it is obvious to bring them together to a new scheduling algorithm called *Periodic Instruction Quantum (PIQ)*. Furthermore, controlling the number of executed instructions is intuitive and pleasant for real-time software developer and an area of active research [6,7].

Similar to the cycle counter algorithm, the time is divided into rounds of constant length. The length of a round is given by a number of *cycles*, not instructions, to keep the temporal length fixed. Within a round, executed instructions are counted for each soft real-time thread separately, i.e. every thread slot has its own counter that is initialized to its specific instruction quantum at the beginning of each round. To decrease the counters when necessary, the *Real-time Issue Stage* notifies in every cycle which thread slot was assigned to which functional unit and the *Priority Controller* updates the counters accordingly.

The real-time threads get fixed priorities in descending order. As soon as a thread (typically the highest priority one) reaches its instruction quantum it is suspended and the priorities of the other threads are effectively risen. From the time when all instruction counters reached zero till the end of the round, only non real-time threads are executed or the sleep mode is activated to save energy.

If a round with an unexpected high computing demand occurs and not all counters could reach zero before the end of the round, the quantum for the next round is added to the remaining counter value, i. e. the unfinished threads get more instructions in the next round. An overflow of the counters ought to be avoided by a schedulability analysis, but throwing an exception might be a practical solution, too.

5.2 Fair Non Real-Time Distribution

So far the remaining non real-time threads only have a fixed priority and are executed accordingly. Therefore the assigned execution time is unequally distributed and declines rapidly with decreasing priorities. A fair (i.e. uniform) distribution would be much more reasonable. To avoid further hardware costs, the instruction counters of the PIQ scheduler can be used to realize a *Round Robin by Instruction Quantum (RRIQ)* scheduling algorithm.

The non real-time threads get an instruction quantum and descending priorities (naturally all lower than the real-time priorities) like the real-time threads, too. Analogous the counters are decreased, if an instruction of the appropriate thread slot is issued. But the limits of the rounds are ignored for the non real-time threads: as soon as a counter reaches zero, the priority controller rises the

priorities of the other non real-time threads and the completed thread is set to the lowest priority but in return gets a new instruction quantum.

Setting the thread to the lowest priority nearly disables it, but every time another thread reaches its instruction quantum, the priority of the former thread is increased by one. Finally, when the thread reaches the highest priority, all other non real-time threads were executed once for quantum instructions since the last time the thread had the highest priority - fairness is granted.

An obvious extension would be to allow individual instruction quanta for every thread slot, to give them different fractions of the total execution time. But the weighting does not lead to the intended result, as the succession of the threads can vary:

If a thread B with second highest priority (of the RRIQ threads) has a considerably lower instruction quantum than thread A with the highest priority, it is possible that thread B's counter reaches zero before thread A and thus obtains the lowest priority. As soon as thread A's counter reaches zero, it follows thread B, gets the lowest priority and thread B hence the second lowest one. After that, thread B has a higher priority than thread A, contrary to the starting point. Thread B has "overtaken" thread A within the round robin queue.

Even with equal quanta the described special case is theoretically possible. Thus the algorithm is not perfect, but in practice it is appropriate to equally distribute the execution time without additional hardware costs.

6 Evaluation

We use 13 benchmarks from EEMBC AutoBech 1.1 [17], see Table 1. These benchmarks consist of an initialization phase that is not part of the benchmarking and a variable number of iterations of the actual benchmark code. Each iteration uses the same input data. We pick iteration counts that result in a single-threaded execution time of at least 1.2 mio cycles. The smallest iteration number is 10 for `idctrn`. All benchmarks were compiled with -O3 optimization of the Hightec GNU C/C++ compiler for TriCore [18].

By combining two benchmarks we constructed 169 task sets, named by two letters, the first indicates the benchmark of the first thread, the second indicates the

Table 1. Used EEMBC benchmarks with low (left row) and high (right row) variation of the local IPC

Abbr	Name	IPC	Fluctuation Range
F	aifirf	0.4584	0.080
H	cacheb	0.4452	0.070
P	pntrch	0.4204	0.065
U	puwmod	0.4487	0.075
R	rspeed	0.4575	0.085
T	ttsprk	0.4227	0.075

Abbr	Name	IPC	Fluctuation Range
A	a2time	0.5122	0.265
B	basefp	0.4652	0.180
M	bitmnp	0.4462	0.235
C	canrdr	0.3844	0.120
D	idctrn	0.5653	0.275
I	iirflt	0.5368	0.350
O	tblook	0.4703	0.280

second thread. There are combinations of the same benchmark (e.g. DD) and all other combinations occur in two variants with different order (e.g. DI and ID).

6.1 Savings Due to Overlapping

The advantages of overlapping threads are evaluated by comparing the soft real-time capable PIQ scheduling (section 5.1) to the hard real-time capable DTS technique (section 3.3). Both algorithms are implemented using a round of 1200 cycles that should be distributed equally to the two threads of a task set. When using DTS, both threads are executed alternately for 600 cycles, when using PIQ each thread gets an individual instruction quantum that matches the number of instructions this thread executes in 600 cycles on average ($= 600 \times AverageIPC$).

To check if both scheduling techniques provide equal progress, each task set is executed with both scheduling algorithms and the termination times of the threads are compared. As the deviation between DTS and PIQ termination time never exceeds 2 rounds (i.e. 2400 cycles or 0.2% of the total execution time), the throughput is considered as being equal. The small differences can be explained by varying release times within a round and rounding errors as the instruction quantum is integer.

When all threads within a round reached their instruction quantum, we call the round *saturated* and the number of cycles from the beginning of the round to the cycle when the round is saturated is called *saturation time*. To estimate the possible savings by overlapping, a task set is scheduled by PIQ and the first 2000 rounds are considered to determine the minimum, maximum and average saturation time.

Fig. 5 shows these three values for every task set. The task sets are plotted from left to right, the letter at the x-axis indicates the benchmark of the first thread. Due to space restrictions, the letter for the second thread is not given, but the 13 task sets within a group of task sets with same primary thread are arranged in the same order as the 13 task set groups themselves.

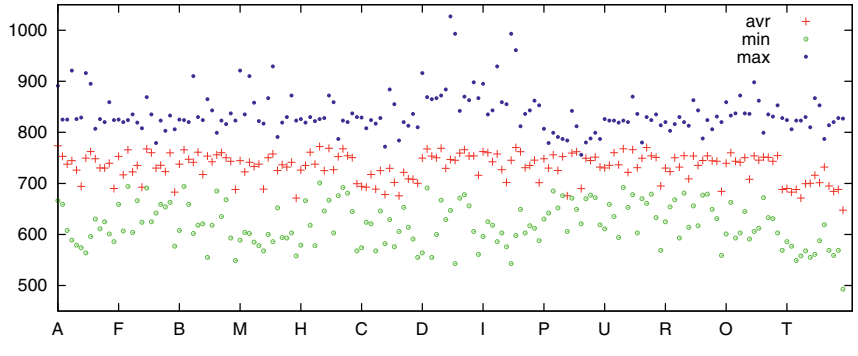


Fig. 5. Minimum, maximum and average saturation time when scheduling two PIQ threads

Table 2. Number of task sets that do not exceed the round boundary (At 50% no task set fails, thus this column gives the total number of task sets in the subsets)

IPC	50%	60%	65%	66%	67%	68%	69%	70%	71%	72%	73%	74%	75%
`lolo`	**36**	**36**	**36**	**36**	**36**	**36**	**36**	**36**	**36**	**36**	**36**	34	25
`hilo`	**42**	**42**	**42**	**42**	**42**	**42**	41	37	37	37	32	28	11
`lohi`	**42**	**42**	**42**	**42**	**42**	**42**	41	37	37	37	32	28	11
`hihi`	**49**	48	45	43	43	39	37	30	23	20	20	14	8
total	**169**	168	165	163	163	159	155	140	133	130	120	104	55

After 1027 cycles every round of every task set is saturated, as Fig. 5 shows. Without the four combinations DD, DI, ID, II even 920 cycles are enough. Therefore only 77% of any round are used, or with other words, the clock rate can be reduced to 77% to save energy. Another possibility would be to put the processor into sleep mode when the threads are satisfied and to resume it by a periodical interrupt at the beginning of the next round. Assuming 10 cycles for suspending and 10 cycles for resuming would yield an average of 37% idle time for each task set (a minimum of 33.9% for AA and a maximum of 44.4% for TT).

6.2 Maximize Instruction Quantum of Two Threads

Another possibility to use the spare cycles at the end of a round, is to increase the instruction quantum beyond 50%, but still give both threads the same percentage. The last row of Table 2 shows the number of task sets that are executed correctly (i.e. every round is saturated before the next round starts) with IPCs of more than 50% of single-threaded IPC.

A closer look at the failing task sets reveals that only task sets consisting of certain benchmarks cause the scheduler to fail. Notable is the high IPC oscillation of these benchmarks (Fig. 8, 9) compared to other benchmarks with smooth IPC distribution (Fig. 6, 7). Therefore we divided the benchmarks into two groups with low and high IPC oscillation. For classification we use the fluctuation range of the *local IPC*. If the difference between highest and lowest local IPC is smaller than 0.1, the benchmark is classified as low oscillating, else as high oscillating.

Determining the local IPC is difficult, because it must be calculated over an interval of several cycles. The number of cycles is not defined, but can extremely influence the result (Fig. 6 - 9 give the local IPC based on 200 or 1200 cycle intervals and the average IPC). We choose an interval of 200 cycles, but even with different intervals the two classes can easily be seperated.

Consequently the task sets can be divided into 4 groups with low oscillation (`lo`) or high oscillation (`hi`) of the first and the second thread. Table 2 shows the result: as expected, `lolo` task sets can be scheduled with higher relative IPCs than the `hihi` task sets and `lohi` and `hilo` are in between with identical results (i.e. it does not matter if the thread with the higher oscillation is the first or the second one).

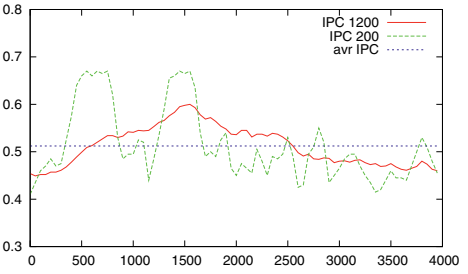


Fig. 6. IPC of a2time (A)

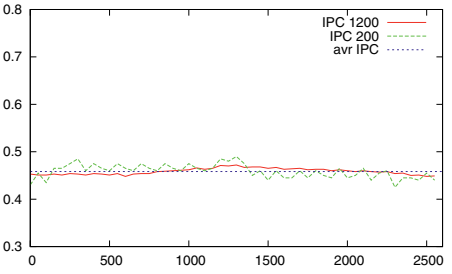


Fig. 7. IPC of aifirf (F)

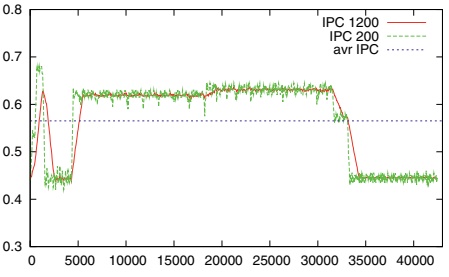


Fig. 8. IPC of idctrn (D)

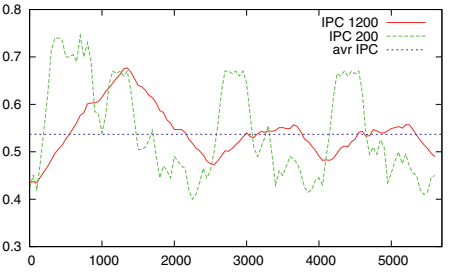


Fig. 9. IPC of iirflt (I)

Instead of granting every thread the same performance share, one thread can get the maximum possible IPC and in return the IPC of the second thread can be reduced to an amount that is still schedulable.

To determine the maximum target IPC, all task sets are executed with 50% for the first thread and 10% for the second thread. Starting from this baseline, the IPC of the first thread is increased until the first task set (DD) missed a round boundary. The previous percentage is the maximum target IPC for the first thread. In the second try the IPC of the first thread is set to this maximum IPC and the percentage of the second thread is increased until a round of any task set is not saturated. Table 3 shows the results when applying this procedure on the whole task set and its four subsets.

6.3 Mixed Real-Time

Finally, we combined DTS and PIQ scheduling to provide one hard real-time thread that runs as if it were single-threaded and two further soft real-time threads with a controlled IPC. We obtain the results similarly to the maximum target IPC in previous section: Initially the percentages of the PIQ threads are set to 10% and increased until the first round violations occur, Table 4 shows the results. The important difference is, that the hard real-time thread is privileged over the PIQ threads and consumes lots of resources.

Table 3. Max. target IPC depending on the task set's IPC fluctuation characteristic

Group	lolo	lohi	hilo	hihi	minimum
First Thread IPC	89%	89%	75%	75%	75%
Second Thread IPC	68%	63%	56%	49%	49%
Total Throughput	157%	152%	131%	124%	124%

Table 4. Maximum IPC for PIQ that are scheduled concurrently to a DTS scheduled hard real-time Thread

DTS Thread	lo				hi			
PIQ Threads	lolo	lohi	hilo	hihi	lolo	lohi	hilo	hihi
Second Thread IPC	57%	57%	55%	54%	61%	59%	47%	47%
Third Thread IPC	29%	21%	15%	21%	27%	16%	30%	24%
Total Throughput	186%	178%	170%	175%	188%	175%	177%	171%

7 Conclusion

We showed, that an in-order SMT processor suits very well the requirements of hard and soft real-time systems. Our architecture supports privileged execution of hard real-time threads, IPC control of soft real-time threads and a fair round robin scheduler for non real-time threads. The IPC control can be used to reduce the energy consumption by 37% or to increase the throughput to at least 124%. Concurrent execution of one hard and two soft real-time threads allows a throughput of more than 170%.

References

1. Siemens VDO: IAA 2007: A New direction in electronics architecture –A Modular concept enables new functions. Press release SV 200709.007 en
2. Mische, J., Uhrig, S., Kluge, F., Ungerer, T.: Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads. In: Proceedings of the 26th IEEE International Conference on Computer Design (October 2008)
3. Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L.: Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In: Proceedings of the 23rd Annual International Symposium on Computer Architecture, pp. 191–202 (May 1996)
4. Raasch, S.E., Reinhardt, S.K.: Applications of Thread Prioritization in SMT Processors. In: Proceedings of the 1999 Workshop on Multithreaded Execution, Architecture, and Compilation (January 1999)
5. Jain, R., Hughes, C.J., Adve, S.V.: Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. In: Proceedings of the 23rd IEEE International Real-Time Systems Symposium, pp. 134–145 (December 2002)
6. Cazorla, F.J., Knijnenburg, P.M., Sakellariou, R., Fernández, E., Ramirez, A., Valero, M.: Predictable Performance in SMT Processors. In: Proceedings of the 1st Conference on Computing Frontiers, pp. 433–443 (April 2004)

7. Yamasaki, N., Magaki, I., Itou, T.: Prioritized SMT Architecture with IPC Control Method for Real-Time Processing. In: Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 12–21 (April 2007)
8. Dorai, G.K., Yeung, D., Choi, S.: Optimizing SMT Processors for High Single-Thread Performance. Journal of Instruction-Level Parallelism 5 (April 2003)
9. Dorai, G.K., Yeung, D.: Transparent Threads: Resource Sharing in SMT Processors for High Single-thread Performance. In: Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques, pp. 30–41 (September 2002)
10. Kreuzinger, J., Schulz, A., Pfeffer, M., Ungerer, T., Brinkschulte, U., Krakowski, C.: Real-time Scheduling on Multithreaded Processors. In: 7th Int. Conference on Real-Time Computing Systems and Applications, pp. 155–159 (December 2000)
11. Brinkschulte, U., Pacher, M.: Implementing Control Algorithms Within a Multithreaded Java Microcontroller. In: Beigl, M., Lukowicz, P. (eds.) ARCS 2005. LNCS, vol. 3432, pp. 33–49. Springer, Heidelberg (2005)
12. El-Haj-Mahmoud, A., AL-Zawawi, A.S., Anantaraman, A., Rotenberg, E.: Virtual Multiprocessor: An Analyzable, High-Performance Architecture for Real-Time Computing. In: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 213–224 (2005)
13. Anantaraman, A., Seth, K., Patil, K., Rotenberg, E., Mueller, F.: Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems. In: ISCA 2003: Proceedings of the 30th annual international symposium on Computer architecture, pp. 350–361 (2003)
14. Infineon Technologies AG: TriCore 1 User's Manual. V1.3.8 (January 2008)
15. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM 20(1), 46–61 (1973)
16. Uhrig, S., Wiese, J.: Jamuth – An IP Processor Core for Embedded Java Real-Time Systems. In: Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems, pp. 230–237 (September 2007)
17. EEMBC: AutoBench 1.1 Software Benchmark Data Book, `http://www.eembc.com/TechLit/Datasheets/autobench_db.pdf`
18. HighTec EDV-Systeme GmbH: Website, `http://www.hightec-rt.com/`

A Hardware Task Scheduler for Embedded Video Processing

Ghiath Al-Kadi and Andrei Sergeevich Terechko

NXP Semiconductors
High Tech Campus 32 (floor 1, office 132),
5656 AE Eindhoven, The Netherlands
{Ghiath.Alkadi,Andrei.Terechko}@nxp.com

Abstract. Modern embedded Systems-on-a-Chip deploy multiple programmable cores to meet increasing performance requirements of video, graphics, and modem applications. However, software implementations of task scheduling and inter-task synchronization often limit performance improvements of multicores. Remarkably, several demanding video applications (e.g. H.264 video decoding) rely on task dependency graphs that can be constructed from a simple dependency pattern. Based on such a pattern, our novel hardware task scheduler can quickly create, order, synchronize and map tasks to cores. We found that our hardware task scheduler speeds up a Quad HD H.264 video decoding by 1.17 times compared to a chip multi-processor with a state-of-the-art hardware task queues. Moreover, our hardware task scheduler allows decreasing the number of cores needed to meet the real-time performance requirements for the H.264 decoder and, consequently, reduces the silicon area of the multicore by up to 12.5%.

Keywords: Hardware task scheduler, task dependency patterns, H.264 video compression, embedded video processing.

1 Introduction

Modern embedded applications such as video, graphics, modems demand higher performance from the same silicon area. To boost performance density embedded chips often deploy domain-specific *multicore* architectures. In many embedded multicore chips each core runs a relatively independent application (e.g. audio, video, 3D graphics), requiring little synchronization and communication with other cores. However, if an application, such as the Quad HD H.264 video decoding, exceeds the compute power of a standalone core, the application may be split into tasks and run on several cores [1].

To run one application on several cores, tasks have to be *created*, *scheduled* in time, *mapped* to proper cores and *synchronized*. These task manipulations strongly depend on the application domain. For example, in web server applications many independent tasks constitute the overall workload and inter-task synchronization is rare. In the embedded video domain, on the other hand, several applications [1][2] can be parallelized using dependent tasks, i.e. a task may start after one or several other tasks have finished. Such applications rely on inter-task dependency graphs.

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 140–152, 2009.

Task scheduling and mapping to cores can be done *statically* at compile-time or *dynamically* at run-time. Modem applications often rely on static schedules to comply with hard real-time deadlines, whereas video decoding has more data-dependent execution time. For example, macroblock Vector Prediction in H.264 video decoding requires between 20 and 1153 cycles [3]. Parallel workload with variable execution times is more amenable to dynamic scheduling to optimize load balancing.

The main focus of our paper is embedded video applications (e.g. H.264 video compression), exhibiting variable execution time and inter-task dependencies. Dynamic run-time scheduling of tasks obviously incurs execution time *overhead*, that according to [3] can reach 114 cycles. For fine-grain tasks of 20 cycles this overhead drastically limits multicore speedups. Accelerating dynamic scheduling, therefore, may enable higher multicore performance. Remarkably, H.264 decoding parallelized at macroblock level [1] has a *repetitive inter-task dependency pattern*, which is sufficient to build a task dependency graph for a complete video frame. Fig. 1 shows a 5x5 macroblocks video frame and two dependencies of one macroblock from the H.264 video decoder. These two relative dependencies repeat for all macroblocks and such repeated dependencies we call a repetitive dependency pattern. Note, that from the pattern we can construct the complete task dependency graph shown on the right in Fig. 1. In contrast to irregular task graphs that are difficult to capture in hardware, the repetitive nature of the inter-task dependencies prompts for hardware acceleration of the task scheduling overhead.

Repetitive dependency patterns can also be found in other *video decoding* algorithms, e.g. the transform coefficient prediction in VC-1 [4]. Furthermore, similar patterns are present in the motion estimation algorithms [2], which are widely used in *video encoding* [5], as well as *video post-processing* such as motion-compensated frame rate up-conversion [6]. Note that although this paper focuses on video applications, repetitive inter-task dependency patterns occur in loop nests of other applications. Such dependencies are called loop-carried dependencies [7], which for certain loop nests exhibit a repetitive nature in the iteration space.

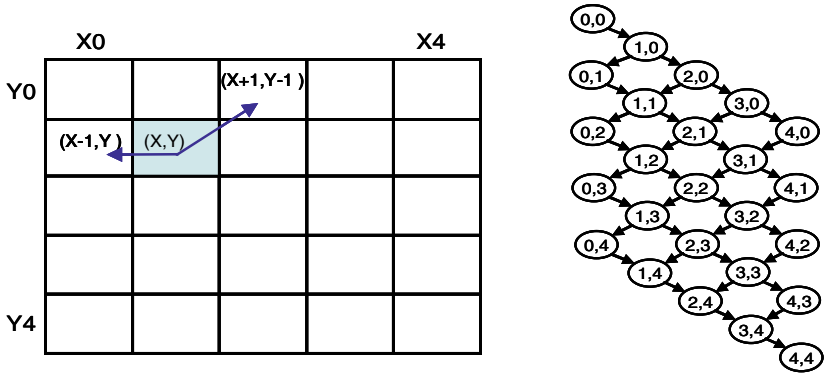


Fig. 1. An H.264 dependency pattern (left), a corresponding task graph (right)

Contributions of our paper include:

1. identification of a *repetitive dependency pattern* as a basis for efficient hardware-accelerated task management;
2. hardware task scheduler (HTS) *architecture*, enabling fine-grain task creation, scheduling, mapping and synchronization;
3. *evaluation* of the multicore performance density gain due to the small HTS reducing the number of cores required to perform H.264 video decoding in real-time.

The remainder of the paper is organized as follows. First, the Related work section outlines state-of-the-art in task scheduling. In Section 3 we detail our novel hardware task scheduler architecture and its programming model. Then, Section 4 describes the experimental setup used for performance evaluations. In Section 5 we discuss the speedup results of H.264 video decoding on up to 16 cores and Section 6 concludes our findings

2 Related Work

We distinguish between dynamic task management in software and hardware.

2.1 Software Task Management

Operating systems, thread libraries and run-time systems provide generic task management support with a fairly large overhead of hundreds to thousands of cycles due to the blocking nature of synchronizations, e.g. in Posix threads and real-time semaphores [8]. Certain run-time systems (e.g. Cell superscalar and SMP superscalar from Barcelona Supercomputer Center [9]) support dynamic task graph creation, while providing simplified programming models. For the embedded domain software task management can reach 114 cycles on a TriMedia core [3]. Although, the title of [10] contains “hardware task scheduler”, the paper describes a task scheduler running in software that schedules tasks on hardware blocks.

Note that *synchronization* for software task management is often also implemented in software, e.g. based on Load Linked (LL) and Store Conditional (SC) [11] machine operations. Such machine operations enable building diverse synchronization primitives (e.g. semaphores or barriers).

2.2 Hardware-Accelerated Task Management

Several research studies evaluated hardware task queues, for example, the Intel CARBON [14], Task Scheduling Unit [12] and the hardware support for multi-core scheduling [13]. In these studies task submission to a particular core is accelerated by hardware task queues, replacing software data structures and the corresponding synchronization overhead. However, inter-task synchronization is still performed in software. Instead of relying on memory-based instructions such as `LL`/`SC`, synchronization (between tasks) can be accelerated in hardware. For example, Cell Broadband Engine [18] introduces hardware mailboxes and semaphores, respectively. The drawback of

most hardware solutions for synchronization is the restricted number of simultaneously active synchronization primitives.

Saez et al. in [15] describe a hardware scheduler accelerating combined scheduling of soft and hard real-time jobs on a uniprocessor. Our scheduler in contrast tackles the task scheduling problem for a multicore, involving complex task-to-core mapping. Furthermore, our task scheduler can create task dependency graphs, whereas the scheduler from [15] obtains a set of independent tasks from the program. Other architectures, such as NVIDIA Tesla [16], are also known to provide hardware acceleration for task scheduling of independent tasks.

Själander et al in [3] propose a programmable *task management unit* (TMU), which similarly to our Hardware Task Scheduler accelerates task creation and synchronization in hardware. TMUs exchange tasks via a centralized task queue. A TMU runs a look-ahead program preparing tasks that will become ready in a short while. However, the programmable nature of the TMU implies a larger silicon area than that of our HTS.

Själander reports that optimized software task manipulations can take about 114 cycles [3], while a generic hardware-accelerated task scheduling can take 15 cycles for task dispatching and around 30 cycles for the synchronizations in software [12]. However, for fine-grain tasks executing of a few tens of cycles, we need even faster task scheduling. In order to achieve lower overhead, we propose to implement task creation, scheduling, mapping and synchronization in a dedicated Hardware Task Scheduler.

3 Hardware Task Scheduler

The main purpose of the Hardware Task Scheduler (HTS) is accelerating task management in hardware. First, the parallel program initializes the HTS with a dependency pattern and loop boundaries sufficient to create a task dependency graph. Then, the HTS prepares available tasks for execution on cores, while gradually building the graph. The cores continuously ask for new tasks and inform the HTS when tasks finish. Finally, the HTS signals to all cores when the task graph is executed.

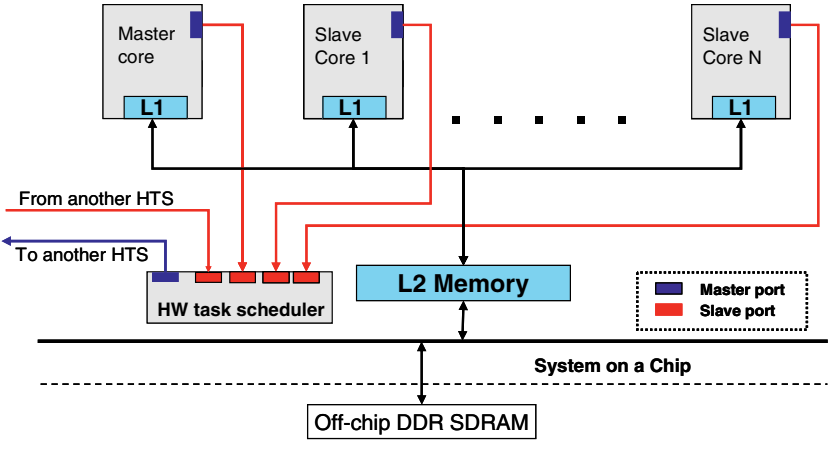


Fig. 2. The HTS in the multi-core system

Our System-on-a-Chip (SoC) consists of several homogeneous cores, shared memory and the HTS, which is connected to each core, see Fig. 2.

The master core configures the HTS with a dependency pattern and loop boundaries, whereas slave cores pull tasks from HTS. Note, that if the number of tasks exceeds the number of available processor cores, a core may execute several tasks sequentially. For large scale systems with several HTSs, each HTS can steal jobs from another HTS using his master interface, see "From another HTS" and "To another HTS" in Fig. 2.

Cores interact with the HTS via a simple API including configuration, get task and release primitives. These primitives are implemented on top of Memory Mapped Input/Output (MMIO) read and write transaction and, therefore, they require no ISA extension and compiler modifications.

3.1 Programming Model

Mapping an application such as H.264 video decoding on a multicore with an HTS requires three steps:

1. parallel kernel isolation;
2. identification of the repetitive inter-task dependency pattern;
3. HTS initialization and kick-off.

In the first step the programmer identifies a kernel applied to different array elements, such as macroblocks, and isolates the kernel in a C function. This parallelization method corresponds to data partitioning, which provides scalable performance with video resolution growth. Interestingly, prior studies proposed automatic parallelization of loops with dependencies similar to H.264 [17]. Applied to our code such parallelizing compilers [22] help eliminate step 2. However, the programmer remains responsible for the code restructuring of step 1. Fig. 3 shows a sequential image filter code, where the macroblock kernel is already isolated in C function `Process_MB()`. For the second step we examine the kernel function in Fig.3 and discover that each `(x,y)` macroblock of the image depends on the value of two macroblocks `(x-1,y)` and `(x+1,y-1)`. Such a dependency pattern is also depicted in Fig. 1.

```
1.  char Image [ImageHeight][ImageWidth];
2.  /*kernel function*/
3.  void Process_MB(int X, int Y){
4.    Image [X][Y] = Image[X-1][Y]+Image[X+1][Y-1];
5.  }
6.  void Process_MB_seq(void){
7.    for (int Y=0; Y < ImageHeight; Y++)
8.      for (int X =0; X < ImageWidth; X++){
9.        …
10.       Process_MB(X, Y);
11.       …
12.     }
13. }
```

Fig. 3. Sequential kernel Process_MB()

In the final third step, the nested loop(s) are removed and replaced by the HTS initialization code, see Fig.4. The initialization code of the HTS includes passing the dependency pattern (`DependencyX` and `DependencyY`), the function pointer (`wrapper_p`), and loop indices (`ImageHeight` and `ImageWidth`) to the HTS. For our example, arrays `DependencyX` and `DependencyY` will contain `(-1,+1)` and `(0,-1)`, respectively. The kernel function is in-lined in a special wrapper HTS function (`wrapper_p`), which requests available tasks from the HTS and starts the kernel with arguments obtained from the HTS. Function `TaskSchdulerInitALL()` will also kick-off the HTS, see line 4. Note line 10, where the master core starts executing the kernel tasks as well.

```
1. void Process_MB_par(void){
2.   wrapper_t wrapper_p = &WRAPPER_Process_MB;
3.   /*configure HTS*/
4.   TaskSchdulerInitALL (wrapper_p,
5.     DependencyX, /*X coord of dependency array*/
6.     DependencyY, /*Y coord of dependency array*/
7.     ImageHeight,
8.     ImageWidth,
9.     Control);    /*start signal for HTS*/
10.  wrapper_p();   /*Start the kernel */
11.}
```

Fig. 4. Parallelized kernel Process_MB() using the HTS

The structure of the wrapper function is shown in Fig. 5. The wrapper reads the HTS status in line 5, and if there is a ready task we start up the kernel with the just read X and Y arguments in line 9. If the HTS reports that the task graph is done then the wrapper returns. Note that the accesses to the HTS `hts_read()` and `hts_write()` are MMIO loads and stores.

```
1.  #define WRAPPER(kernel_fun)
2.    void WRAPPER_##kernel_fun(void){
3.      kernel_t kernel_fun_p = &kernel_fun;
4.      while (1) {
5.        Status = hts_read(HTS_STATUS);
6.        X = hts_read(HTS_X);
7.        Y = hts_read(HTS_Y);
8.        if (Status == HTS_GETTASKSTART) {
9.          kernel_fun_p(X,Y);
10.         hts_write(HTS_CURTASKDONE);
11.       } else if (Status == HTS_TASKGRAPHISDONE)
12.           return;
13.     }
14.   }
```

Fig. 5. HTS wrapper function

Initially, the boot loader of the multicore system triggers the slave cores to load the `slave` program from Fig. 6. Then, the `slave` program requests a function pointer to the parallelized kernel every time the HTS starts.

```
1.  void slave(void) {
2.    int val1, start=0;
3.    while (1) {/*wait until the HTS is started*/
4.    start = hts_read(HTS_CONTROLADDR);
5.    if (start!=0) {
6.      /*Read the function pointer from the HTS*/
7.      TaskSchedulerInitRead(&val);
8.      /*Start the kernel on the slave core(s)*/
9.      ((wrapper_t) val)();
10.     }
11.   }
12. }
```

Fig. 6. The slave program

Note, that the kernel's instruction code is part of the master core's program, but thanks to shared memory the slave cores can easily jump to the loaded code compiled for the master core. When the kernel finishes, all slave cores return back to the slave function and wait for a next kernel.

3.2 Architecture

Fig. 7 shows the top-level architecture components of the HTS:

- **slave0** to **slaveN** slave ports connect to all cores and another external HTS acting as a core;
- **Master Port** allows task stealing from another HTS;
- **Control Unit** orchestrates overall HTS operation;
- **Floating Ready Task FIFO** holds tasks ready for execution but not assigned to a slave port;
- **Slave Ready Task FIFOs** contain tasks ready for execution and assigned to slave ports;
- **Slave Candidate Buffers** stores task candidates of an ongoing task;
- **Synchronization Buffer** stores the coordinates of the last finished tasks;
- **Repetitive Dependency Pattern Buffer** holds the `X` and `Y` relative coordinates of the repetitive dependency pattern.

Let us detail a typical HTS operation sequence. First, the master core executes a sequential part of application and encounters a parallel segment programmed for the HTS. Then, the master core writes a kernel's function pointer, a repetitive dependency pattern and boundaries of the task graph to the HTS. The Control Unit stores the dependency pattern in the Repetitive Dependency Pattern Buffer and submits the first task to a Slave Ready Task FIFO. In parallel, the Control Unit computes all possible candidates that could start executing once the first task is completed and stores them

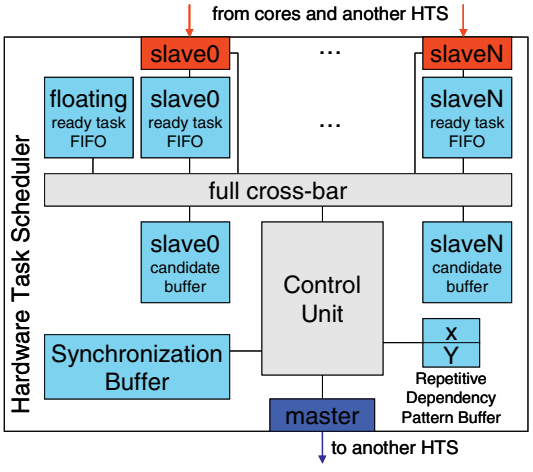


Fig. 7. Hardware Task Scheduler organization

in the proper Slave Candidate FIFO. The candidate calculation is performed by negating the dependency pattern. For example, dependency `(-1,+2)` will be generate task candidate `(x+1,y-2)` relatively to executing task `(x,y)`. Typically, a task generates as many candidates as there are dependencies in the pattern, however, tasks located on the boundary of the task graph may generate one or zero candidates. Hence, a new task for the dependency pattern from Fig. 3 generates two candidates and each candidate has two tasks that it will depend on.

When a Slave interface signals the finish of a parent task execution, the Control Unit tries to promote candidate tasks from the corresponding Slave Candidate Buffer to a Slave Ready Task FIFO. The Control Unit reads one candidate per cycle and checks the following two conditions:

1. if all parent tasks of the candidate have been completed;
2. if no other duplicate candidate task was promoted. Note, that candidate duplicates may be generated for a task depending on several other tasks.

The two checks rely on task statuses captured in the Synchronization Buffer. A task status can be UNPROCESSED, PENDING, or FINISHED. In the beginning all tasks are UNPROCESSED. When a task is promoted to a Slave Ready Task FIFO, its status changes to PENDING. Finally, when a task completes, its status becomes FINISHED. The first check is satisfied, if our candidate's parents statuses are FINISHED. The second check is true, if our candidate's status is UNPROCESSED. If both checks are satisfied, then the candidate migrates to a Slave Ready Task FIFO and updates its status in the Synchronization Buffer. Otherwise, the Control Unit drops the candidate, because another unfinished parent task will promote this task in future or our candidate was a duplicate. The Synchronization Buffer is controlled by the centralized Control Unit, which orders all accesses to the buffer from multiple cores.

Using the full cross-bar in Fig. 7 the Control Unit can map a candidate task to any of the Slave Ready FIFOs. The task-to-core mapping of the HTS is based on the Tail Submit principle from [12]. The Tail Submit improves data cache locality, by

mapping the child task to the parent core, which may already contain some data in the data cache required for the child task. If several child tasks are ready for execution after the finish of the parent task, we map the task on the same Y axis to the parent core and the other child tasks to the Floating Ready Task FIFO. Any idle processor can take a task from the floating FIFO, if its dedicated Slave Ready Task FIFO is empty, which may occur, for example, at the boundaries of the task graph.

3.3 Implementation

Table 1 shows estimates of area and power for various HTS configurations obtained by Register Transfer Level (RTL) synthesis of large blocks of the HTS for the CMOS 45 nm worst-case TSMC library. The HTS grows with the number of cores due to the growth of core interfaces and task FIFOs per core. Furthermore, the application influences the size of the HTS by the required size of the synchronization buffer. In general, the area and power of the HTS are smaller than 5% of the TriMedia TM3270 core [21].

Table 1. Area and power estimates for various HTSs

		1 core	2 cores	4 cores	8 cores	16 cores
SD 720x560	Area (mm^2)	0.032	0.037	0.048	0.069	0.109
	Power (mW/MHz)	0.034	0.040	0.052	0.076	0.124
HD 1920x1080	Area (mm^2)	0.034	0.039	0.050	0.071	0.111
	Power (mW/MHz)	0.034	0.040	0.052	0.076	0.124
QHD 3840x2160	Area (mm^2)	0.037	0.042	0.053	0.074	0.114
	Power (mW/MHz)	0.035	0.041	0.053	0.077	0.125

4 Evaluation Framework

The HTS was modeled in the TTIsim simulator, which is part of the NXP TriMedia Compilation System v5.1 [19]. We extended the TTIsim to instantiate multiple TriMedia TM3270 VLIW cores [22]. Each core has a 64 KB data cache coupled to other cores' caches and the shared memory via a MESI cache coherence protocol. The off-chip SDRAM is not modeled, however, a 40 cycle L1 refill latency is simulated.

We evaluated HTS performance on the H.264 video decoding, which represents modern demanding video processing applications. The production quality H.264 codec written in C++ was highly optimized for the TriMedia TM3270 processor and was task-level parallelized as described in [12]. Our evaluations exclude the CABAC entropy decoding of the H.264 video decoding, which is likely to be implemented in a dedicated hardware accelerator synchronizing with the cores using interrupts. We adapted the parallel H.264 codec with the HTS API described in Section 0. The video input sequences are taken from the SVT high definition Multi Format test set [20], available from Sveriges Television, and the Taurus Media Technik collection. The sequences are called "CrowdRun" and "Tractor" with the following resolutions: Quad HD (QHD) 3840x2160 and Full HD (FHD) 1920x1080, respectively. The H264 encode/decode profile used is the main profile. We simulated 25 frames from each

sequence. The H.264 application and the API were compiled with the production compiler TriMedia C/C++ compiler v5.1 and executed on the TTIsim simulator.

We benchmarked the HTS against the hardware Task Scheduling Unit [12], based on the CARBON concept [14]. Furthermore, we compared the HTS versus the software implementation, which uses the Task Pool (TP) model [12]. Both the TSU and Task Pool measurements were performed with the Tail Submit software optimization and without [12]. The Tail Submit optimization avoids task creation overhead by executing a new task within the already running task.

5 Experimental Result

The simulation results are shown in Fig. 8 to 12. Fig. 8 and 9 shows the speedups of H.264 decoding versus the number of cores for the Full HD input stream. Fig. 10 and 11 present results for the Quad HD input stream. The results clearly demonstrate that the HTS outperforms both the software and the state-of-the-art hardware implementations of task scheduling for all input streams. The HTS achieves this performance thanks to the low overhead of 15 cycles to create, schedule, map and synchronize tasks given a task dependency pattern. The mentioned cycle count includes 5 cycles of the internal operation of the HTS, 4 cycles in the clock domain bridges and, finally, 6 cycles are spend in the application software accessing the HTS.

The HTS enables relatively higher speedups for the MESI cache coherent memory, rather than for the idealized perfect memory without caches in Fig. 9 and 11. The speedup of HTS relative to TSU and TP using the MESI cache is 1.17 times, while the measured relative speedup using idealized perfect memory without caches is 1.05 times. This can be attributed to efficient data locality scheduling of the HTS. Remarkably, the Tail Submit optimization accounts for 26% and 14% performance improvement for TP and TSU, respectively, see Fig. 8. Although, the HTS does not profit from this optimization, it still outperforms the TSU and TP. Furthermore, by adding more cores to the system, the HTS shows higher scalability than the other approaches.

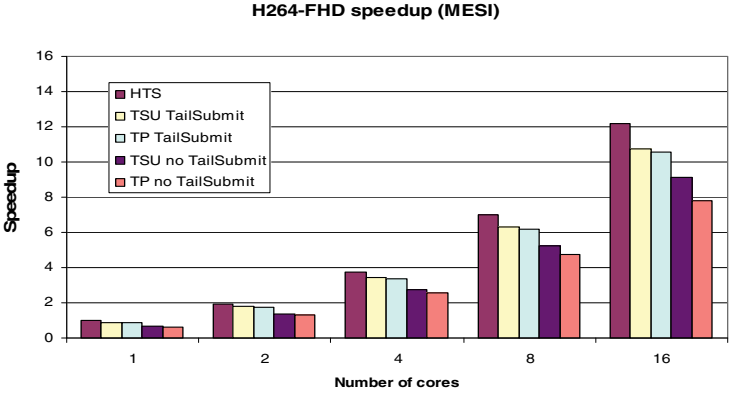


Fig. 8. H.264 Full HD speedup using caches with MESI

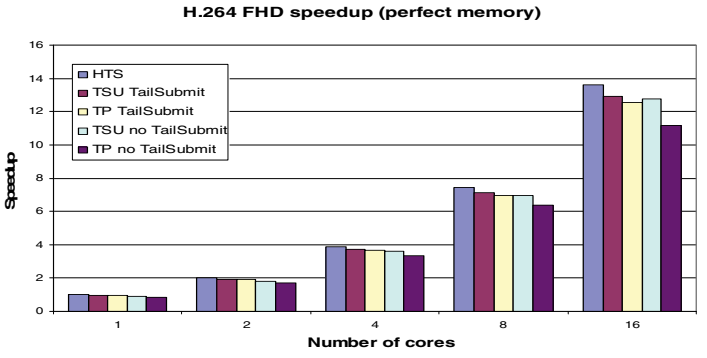


Fig. 9. H.264 Full HD speedup using caches with perfect memory

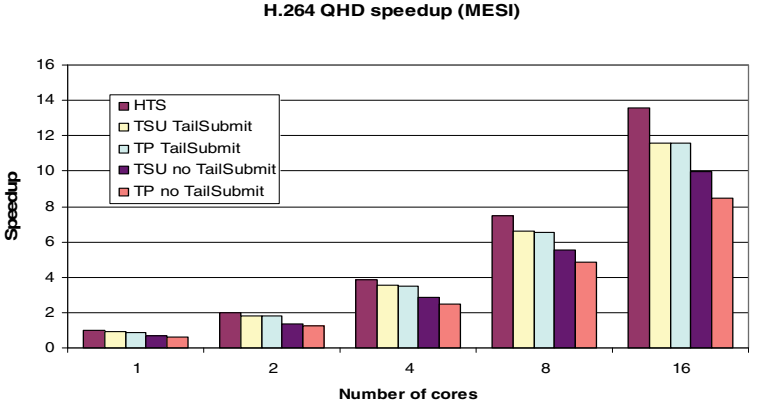


Fig. 10. H.264 Quad HD speedup using caches with MESI

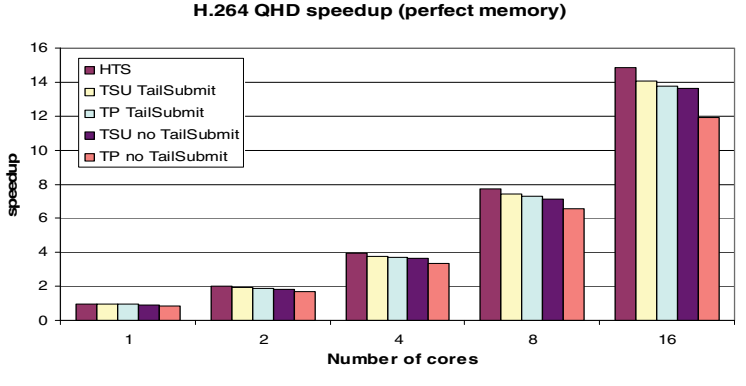


Fig. 11. H.264 Quad HD speedup using caches with perfect memory

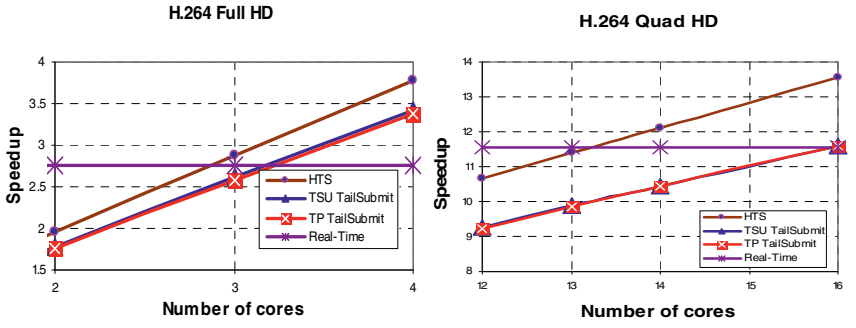


Fig. 12. Number of cores required to meet the real time, Full HD (left) and Quad HD (right)

As shown in Fig. 12, the required number of cores to meet real-time with the HTS is 3 for Full HD and 14 for Quad HD H.264 video decoding. On the other hand, the optimized software implementation as well as the TSU require 4 and 16 cores for Full HD and Quad HD, respectively. The area reduction thanks to the HTS can reach 25% and 12.5% for Full HD and Quad HD respectively, which is very important for low-cost embedded systems. However, the speedups of the HTS, TP, and TSU for real-time Full HD are very close, therefore, we can claim area reduction only for Quad HD.

6 Conclusion

Based on a task dependency pattern, the hardware task scheduler can quickly create tasks, schedule them in time, synchronize and map tasks to cores. In contrast to hundreds of cycles spent on task scheduling in software, the hardware task scheduler requires only 15 cycles for both scheduling and synchronization. In general, the HTS speeds up the optimized software implementation of task scheduling and synchronization by 1.173 times for parallelized Quad HD H.264 video decoding on a 16 cores processor. Moreover, due to the integrated acceleration of inter-task synchronization, the hardware task scheduler outperforms the state-of-the-art CARBON hardware task queues by 1.169 times. Furthermore, the hardware task scheduler allows decreasing the number of cores required to meet the real time for the H.264 decoder and, consequently, reduces the silicon area by 12.5%.

Acknowledgements

We thank Jan Hoogerbrugge for the Task Scheduling Unit simulator model and Marc Duranton for inspiring discussions.

References

1. van der Tol, E.B., Jaspers, E.G., Gelderblom, R.H.: Mapping of H.264 Decoding on a Multiprocessor Architecture. In: Proceedings of SPIE Conference on Image and Video Communications and Processing, pp. 707–718 (2003)

2. de Haan, G., Biezen, P.W.A.C., Huijgen, H., Ojo, O.A.: True-motion estimation with 3-D recursive search block matching. IEEE Transactions on Circuits and Systems for Video Technology 3, 368–379 (1993)
3. Själander, M., Terechko, A., Duranton, M.: A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures. In: 11th EUROMICRO Conference on Digital System Design (DSD), Parma, Italy, September 3rd - 5th (2008)
4. Lakshmish, R., Praveen, K.Y., Maiti, K., Pai, J., Adhikary, T.K.: Efficient Implementation of VC-1 Decoder on Texas Instrument's OMAP2420 – IVA. In: 14th International Workshop on Systems, Signals and Image Processing, 2007 and 6th EURASIP Conference focused on Speech and Image Processing, Multimedia Communications and Services, pp. 430–433 (June 2007)
5. Fluegel, S., Klussmann, H., Pirsch, P., Schulz, M., Cisse, M., Gehrke, W.: A Highly Parallel Sub-Pel Accurate Motion Estimator for H.264. In: IEEE 8th Workshop on Multimedia Signal Processing, October 2006, pp. 387–390 (2006)
6. van de Waerdt, J.-W., Vassiliadis, S., Bellers, E., Janssen, J.G.W.M.: Motion estimation and temporal up-conversion on the TM3270 media-processor. In: International Conference on Consumer Electronics, January 2006, pp. 315–316 (2006)
7. Srikant, Y.N., Shankar, P.: The Compiler Design Handbook: Optimizations & Machine Code Generation. CRC Publisher (2002) ISBN 978-0849312403
8. Mesa, M.A., Ramirez, A., Ayguadey, E., Martorelly, X., Valero, M.: Scalability of Macroblock-level Parallelism for H.264 Decoding. In: Advanced Computer Architecture and Compilation for Embedded Systems, ACACES 2008, Poster Session, L'Aquila, Italy (August 2008)
9. SMP superscalar run-time system, Barcelona Supercomputer Center, `http://www.bsc.es/plantillaG.php?cat_id=385`
10. Kwok, T.T.-O., Kwok, Y.-K.: Practical design of a computation and energy efficient hardware task scheduler in embedded reconfigurable computing systems. In: 20th International Parallel and Distributed Processing Symposium, April 25-29 (2006)
11. Sweetman, D.: See MIPS run. Morgan Kaufmann, San Francisco ISBN 978-1558604100
12. Hoogerbrugge, J., Terechko, A.: A multithreaded multicore system for embedded media processing. Transactions on High-Performance Embedded Architectures and Compilers 3(2) (2008)
13. Lin, Y.: Realizing Software Defined Radio – A Study in Designing Mobile Supercomputers, PhD dissertation, TU/Michigan, Ann Arbor, USA (2008)
14. Kumar, S., Hughes, C.J., Nguyen, A.: Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In: International Symposium on Computer Architecture (2007)
15. Saez, S., Vila, J., Crespo, A., Garcia, A.: A hardware scheduler for complex real-time systems. In: Proceedings of the IEEE International Symposium on Industrial Electronics, 12-16 July 1999, vol. 1, pp. 43–48 (1999)
16. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro. 28(2) (2008)
17. Krishnamoorthy, S., et al.: Effective automatic parallelization of stencil computations. In: Proceeding of the ACM PLDI conference. 42(6), June 2007, pp. 235–240.
18. Cell Broadband Engine, `http://www-128.ibm.com/developerworks/power/cell/`
19. TriMedia Compilation System v5.1, `http://www.nxp.com/`
20. Sveriges Television: The SVT High Definition Multi Format Test Set (2006)
21. van de Waerdt, J.W.: The TM3270 media-processor, PhD dissertation TU/Delft, The Netherlands (2006) ISBN 90-9021060-1
22. Turjan, A.: Compiling Nested Loop Programs to Process Networks, PhD thesis, University Leiden, The Netherlands (2007) ISBN 978-90-9021676-8

Finding Stress Patterns in Microprocessor Workloads

Frederik Vandeputte and Lieven Eeckhout

Department of Electronics and Information Systems (ELIS), Ghent University, Belgium
{fgvdeput,leeckhou}@elis.UGent.be

Abstract. Power consumption has emerged as a key design concern across the entire computing range, from low-end embedded systems to high-end supercomputers. Understanding the power characteristics of a microprocessor under design requires a careful study using a variety of workloads. These workloads range from benchmarks that represent typical behavior up to hand-tuned stress benchmarks (so called stressmarks) that stress the microprocessor to its extreme power consumption.

This paper closes the gap between these two extremes by studying techniques for the automated identification of stress patterns (worst-case application behaviors) in typical workloads. For doing so, we borrow from sampled simulation theory and we provide two key insights. First, although representative sampling is slightly less effective in characterizing average behavior than statistical sampling, it is substantially more effective in finding stress patterns. Second, we find that threshold clustering is a better alternative than k-means clustering, which is typically used in representative sampling, for finding stress patterns. Overall, we can identify extreme energy and power behaviors in microprocessor workloads with a three orders of magnitude speedup with an error of a few percent on average.

1 Introduction

Energy, power, power density, thermal hotspots, voltage variation, and related design concerns have emerged as first-class microprocessor design issues over the past few years. And this is the case across the entire computing range, from low-end embedded systems to high-end supercomputers. A detailed understanding of these issues is of primary importance for designing energy-aware, power-aware and thermal-aware microprocessors, their power and thermal management strategies, their power supply unit, and thermal package.

Understanding the power, energy and thermal characteristics of a microprocessor under design requires appropriate benchmarking and simulation methodologies. At the one end of the spectrum, researchers and engineers consider average workload behavior. This is appropriate for studying a microprocessor's average power consumption or thermal map, however, it does not capture more extreme behaviors. At the other end of the spectrum, stressmarks are being used to explore a microprocessor's maximum power consumption [9,10], maximum thermal hotspots [27], and maximum dI/dt behavior [16]. These stressmarks are typically hand-tuned, and push the microprocessor to its extremes in order to understand the microprocessor's worst-case behavior. These stress patterns are not expected to occur during typical operation, however, they *can* occur and therefore the microprocessor should be able to cope with them.

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 153–167, 2009.

Microprocessors designed for maximum possible power consumption are not cost-effective though because of the large gap between maximum and typical power consumption. Dynamic thermal management (DTM) techniques [1,23] seek to exploit this gap: the microprocessor cooling apparatus is designed for a wattage less than the maximum power consumption, and a dynamic emergency procedure guarantees that this designed-for wattage level is never exceeded with minimal impact on overall performance. Gunther et al. [11] report that DTM techniques based on clock gating permitted a 20% reduction in the thermal design power for the Intel Pentium 4 processor. Developing and evaluating DTM mechanisms however requires adequate evaluation methodologies for quickly finding the extreme behaviors in typical workloads that are subject to DTM.

Therefore, this paper closes the gap between the two ends of the power benchmarking spectrum by studying ways of identifying *stress patterns* in typical workloads, also called 'worst-case execution behaviors' by Tiwari et al. [25]. More specifically, the goal of this work is to find stress patterns in typical workloads with the least possible simulation time. Identifying stress patterns in typical workloads is important because these stress patterns are expected to occur regularly in practice, much more often than the stress patterns represented by hand-tuned stressmarks. The stress patterns are the execution behaviors that DTM emergency procedures should adequately deal with.

We build on sampled simulation theory for identifying stress patterns in typical workloads. However, in contrast to sampled simulation for which the aim is to estimate *average* performance or power consumption by simulating a representative sample of the entire program execution, the goal in this paper is to leverage sampled simulation theory to find a sample of real program execution that includes stress patterns with *extreme* workload behavior, e.g., max power, max energy, etc. There are two common ways in sampled simulation, statistical sampling (as done in SMARTS [29]) and representative sampling (as done in SimPoint [22]). Our experimental results using the SPEC CPU2000 benchmarks confirm that statistical sampling is generally more accurate than representative sampling for estimating average behavior as shown in prior work [30], however, the new insight provided in this paper is that representative sampling is substantially more effective in identifying stress patterns in typical workloads. The intuitive explanation is that representative sampling uses knowledge about the program structure and execution to find representative sampling units, whereas statistical sampling is largely agnostic to any notion of program structure and execution. Sampling units selected through representative sampling therefore have a higher likelihood of including extreme workload behaviors. In addition, we find that threshold clustering is a better clustering method than k-means clustering (which is commonly used in representative sampling such as SimPoint) for identifying sampling units with extreme workload behavior. The end result is that we can estimate stress patterns in typical workloads with a three orders of magnitude simulation speedup compared to detailed simulation of entire workloads with an error of at most a few percent on average.

In this paper, we make the following contributions:

- We close the gap between sampled simulation focusing on average workload behavior and hand-crafted stressmarks focusing on extreme behavior by identifying stress patterns in typical workloads.

- We make the case that representative sampling is substantially more effective in finding extreme behaviors in microprocessor workloads than statistical sampling, although statistical sampling is (slightly) more effective in capturing average behavior.
- The results in this paper motivate changing current simulation practice. Not only does representative sampling using threshold clustering estimate average performance and power nearly as accurate as statistical sampling, it is substantially more accurate when it comes to estimating stress patterns. And although representative sampling may be more commonly used than statistical sampling in current simulation practice, this paper shows that threshold clustering is substantially more effective than k-means clustering (which is typically being used) for finding stress patterns. In other words, representative sampling with threshold clustering is both effective at estimating average performance as well as stress patterns, whereas prevalent techniques (representative sampling with k-means clustering and statistical sampling) are only effective for estimating average performance.
- We show that the proposed method can be used for finding many different flavors of extreme workload behaviors, such as high cache miss rate, low IPC, or low branch predictability behaviors. These behaviors may be useful for understanding program patterns that lead to these extremities.

We believe this work is timely as power is a primary design concern in today's computer systems, and we are in need for appropriate benchmarking and performance analysis methodologies. In addition, stress patterns will become even more relevant as we enter the multi-core era and the gap between average and peak power widens as the number of cores increases. Benchmarking consortia have also recognized the need for energy- and power-oriented benchmarks and associated benchmarking methodologies. For example, SPEC has developed the SPECpower_ssj2008 benchmark suite [24], which evaluates the performance and power characteristics of volume server class computers. Likewise, EEMBC has released the EnergyBench benchmark suite, which reports energy consumption while running performance benchmarks [18].

2 Sampled Simulation

In sampled simulation, only a limited number of *sampling units* from a complete benchmark execution are simulated in full detail. We refer to the selected sampling units collectively as the *sample*. Sampled simulation only reports performance for the instructions in the sampling units, and discards the instructions in the pre-sampling units. And this is where the dramatic performance improvement comes from: only the sampling units, which account for only a small fraction of the total dynamic instruction count, are simulated in a cycle-by-cycle manner.

There are three major issues with sampling: (i) what sampling units to select, (ii) how to initialize a sampling unit's architecture starting image, and (iii) how to accurately estimate a sampling unit's microarchitecture starting image. This paper only concerns the first issue because the other two issues can be handled easily by leveraging existing technology. For example, the architecture starting image (registers and memory state) can be set through fastforwarding or through checkpointing [26,28]; and the

microarchitecture starting image (caches, branch predictors, etc.) can be estimated with microarchitecture state warmup techniques — there is a wealth of literature covering this area, see for example [5,8,12,19,26,28,29].

There are basically two major ways for determining what sampling units to select, namely (i) statistical sampling, and (ii) representative sampling. We now discuss both approaches.

2.1 Statistical Sampling

Statistical sampling takes a number of sampling units across the whole execution of the program. These sampling units are chosen randomly or periodically in an attempt to provide a representative cross-cut of the entire program execution.

Laha et al. [20] propose statistical sampling for evaluating cache performance. They select multiple sampling units by randomly picking intervals of execution.

Conte et al. [5] pioneered the use of statistical sampling in processor simulation. They made a distinction between sampling bias and non-sampling bias. Non-sampling bias results from improperly constructing the microarchitecture starting image prior to each sampling unit. Sampling bias refers to how accurate the sample is with respect to the overall average. Sampling bias is fundamental to the selection of sampling units.

The SMARTS (Sampling Microarchitecture Simulation) approach by Wunderlich et al. [29] proposes *systematic sampling*, which selects sampling units periodically across the entire program execution, i.e., the pre-sampling unit size is fixed, as opposed to random sampling. The potential pitfall of systematic or periodic sampling compared to random sampling is that the sampling units may give a skewed view in case the periodicity present in the program execution under measurement equals the sampling periodicity or its higher harmonics. This does not seem to be a concern in practice though as SMARTS achieves highly accurate performance estimates compared to detailed entire-program simulation. The important asset of statistical sampling compared to representative sampling, is that it builds on well-founded statistics theory, which enables computing confidence bounds at a given confidence level.

2.2 Representative Sampling

Representative sampling contrasts with statistical sampling in that it first analyzes the program execution to pick a representative sampling unit for each unique behavior. The most well known representative sampling approach is the SimPoint approach proposed by Sherwood et al. [22]. SimPoint picks a small number of sampling units that accurately create a representation of the complete execution of the program. To do so, they break an entire program execution into intervals — an *interval* is a contiguous sequence of instructions from the dynamic instruction stream — and for each interval they create a code signature. The code signature is a so called Basic Block Vector (BBV) that counts the number of times each basic block is executed in the interval, weighted with the number of instructions per basic block. After normalizing the BBVs so that the BBV elements sum up to one, they then perform clustering to group intervals with similar code signatures (BBVs) into so called *phases*. BBV similarity is quantified by computing the Manhattan distance between two BBVs. The intuitive notion is that

intervals of execution with similar code signatures have similar architectural behavior, and this has been shown to be the case by Lau et al. [21]. Therefore, only one interval from each phase needs to be simulated in order to recreate an accurate picture of the entire program execution. They then choose a representative sampling unit from each phase and perform detailed simulation on that representative unit. Taken together, these sampling units (along with their respective weights) represent the complete execution of a program. A sampling unit is called a *simulation point* in SimPoint terminology, and each simulation point is an interval with on the order of millions, or tens to hundreds of millions of instructions. The simulation points can be used across microarchitectures because the BBVs, based on which the simulation points are identified, are microarchitecture-independent.

The clustering step in the SimPoint approach is a crucial step as it classifies intervals into phases, with each phase representing distinct program behavior. There exist a number of clustering algorithms; here, we discuss k-means clustering (which is used by SimPoint) and threshold clustering (which we advocate in this paper for identifying stress patterns in typical workloads).

K-means clustering. K-means clustering produces exactly k clusters and works as follows. Initially, k cluster centers are randomly chosen. In each iteration, the distance is calculated for each interval to the center of each cluster, and the interval is assigned to its closest cluster. Subsequently, new cluster centers are computed based on the new cluster memberships. This algorithm is iterated until no more changes are observed in the cluster memberships. It is well known that the result of k-means clustering is dependent on the choice of the initial cluster centers. Therefore, SimPoint considers multiple randomly chosen cluster centers and uses the Bayesian Information Criterion (BIC) [22] to assess the quality of the clustering: the clustering with the highest BIC score is selected.

Threshold clustering. Classifying intervals into phases using threshold clustering can be done in two ways, using an iterative algorithm or using a non-iterative algorithm. The iterative algorithm selects an instruction interval as a cluster center and then computes the distance to all the other instruction intervals. If the distance measure is smaller than a given threshold θ, the instruction interval is considered to be part of that cluster. Out of all remaining instruction intervals (not part of previously formed clusters), another interval is selected randomly as a cluster center and the above process is repeated. This iterative process continues until all instruction intervals are assigned to a cluster/phase. The θ threshold is expressed as a percentage of the maximum possible Manhattan distance between two intervals; the maximum Manhattan distance between two intervals is 2 assuming normalized BBVs, i.e., the sum across all BBV elements equals one.

The non-iterative algorithm scans all intervals from the beginning until the end of the dynamic instruction stream. If the interval is further away from any previously seen cluster center than a given threshold θ, the interval is considered the center of a new cluster. If not, the interval is assigned to the closest cluster. The non-iterative algorithm is computationally more efficient and performs well for our purpose — we therefore use the non-iterative approach in this paper.

The important advantage of threshold clustering is that, by construction, it builds phases for which its in-phase variability (in terms of BBV behavior) is limited to a threshold θ. This is not the case for k-means clustering: the variability within a phase can vary across phases.

3 Experimental Setup

3.1 Benchmarks and Simulators

We use the SPEC CPU2000 benchmarks and all of their reference inputs in our experimental setup. These benchmarks were compiled and optimized for the Alpha ISA; the binaries were taken from the SimpleScalar website; all benchmarks are run to completion.

We use the SimpleScalar/Alpha v3.0 [3] superscalar out-of-order processor simulator. The processor model is configured along the lines of a typical four-wide superscalar microprocessor such as the Alpha EV7 (21364). Power is estimated using Wattch v1.02 [2] and HotLeakage [23] assuming a 70nm technology, 5.6GHz clock frequency and 1V supply voltage. We assume an aggressive clock gating mechanism.

3.2 Sampled Simulation

For statistical sampling, we use periodic sampling, as done in SMARTS [29], i.e., we select a sampling unit every n intervals. We will vary the sampling rate $1/n$ in the results presented in this paper.

For representative sampling, we use SimPoint v3.0 with its default settings. In short, SimPoint computes a BBV per interval, and subsequently performs k-means clustering on randomly projected 15-dimensional BBVs; SimPoint evaluates all values of k between 1 and maxK and picks the best k and random seed per k based on the BIC score of the clustering. We will vary the sampling rate by varying the SimPoint maxK parameter. In the evaluation section of this paper, we will compare k-means clustering versus threshold clustering. For doing so, we replace the k-means clustering algorithm with the threshold clustering algorithm while leaving the rest of the SimPoint software untouched.

In this paper, for both statistical and representative sampling, the interval size is set to 1M (2^{20}) instructions unless mentioned otherwise, i.e., the stress patterns constitute of 1M dynamically executed instructions. This choice does not affect the general conclusions in this paper though — the methodology can be applied to other interval granularities as well. In fact, we experiment with larger interval sizes — not reported here because of space constraints — and obtain similar results as for the 1M-instruction interval granularity. However, for smaller interval granularities, there may be practical considerations that prohibit the use of representative sampling, the reason being that the clustering algorithm may become very time-consuming for a large number of intervals. Addressing the computational concerns of clustering large data sets is left for future work.

4 Evaluation

In the evaluation section, we now compare statistical sampling against representative sampling for finding stress patterns in microprocessor workloads. This is done in a number of steps: we present per-benchmark max power stress patterns, as well as processor component power stress patterns; we also evaluate the error versus simulation speedup trade-off; and finally, we demonstrate the efficacy of the proposed technique for finding other flavors of extreme behavior, such as max CPI, max cache miss rate and max branch misprediction rate stress patterns.

4.1 Motivation

Before evaluating sampled simulation for identifying stress patterns in typical microprocessor workloads, we first further motivate the problem by showing that the variability over time in power consumption is significant within a single benchmark execution. We therefore compute the power consumption on an interval basis, i.e., we compute the power consumption per interval of 1M instructions in the dynamic instruction stream. This yields a distribution of power consumption numbers. Figure 1 represents this distribution as a boxplot per benchmark. The box represents the 5% and 95% quartiles, i.e., 90% of the data lies between these two markers, and thick horizontal line in the box represents the median power consumption across the entire program execution. The outliers are represented by the dashed lines that fall out of the box; the minimum and maximum values are represented by the bottom and top horizontal lines at the ends of the dashed lines, respectively.

The box plots clearly show that there is significant variability over time in power consumption, and, more importantly within the context of this paper, there is a large discrepancy in median versus max power consumption. In fact, for many benchmarks, the max power consumption is substantially higher than its median power consumption, e.g., for mcf the max power consumption is more than three times as high as its median power consumption. And in addition, the bulk of the power consumption numbers falls

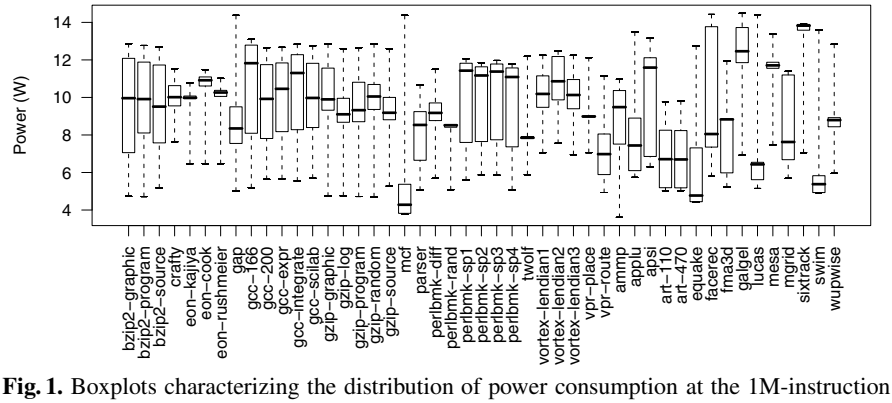


Fig. 1. Boxplots characterizing the distribution of power consumption at the 1M-instruction interval granularity; the boxes represent the 5% and 95% quartiles, and the thick horizontal line in each box represents the median

far below the max power consumption. This illustrates that finding stress patterns for these benchmarks is challenging, i.e., we need to find one of the few intervals that cause max power consumption out of the numerous intervals that constitute the entire benchmark execution — there are typically tens or even hundreds of thousands of 1M-instruction intervals per benchmark.

4.2 Per-Benchmark Stress Patterns

We now evaluate the efficacy of sampled simulation in finding stress patterns at the 1M-instruction interval granularity. For doing so, we assume a $1000\times$ simulation speedup for both statistical and representative sampling compared to the simulation of the entire program execution; we will consider other simulation speedups in Section 4.4. Simulation speedup in this paper is defined as the number of instructions in the entire benchmark execution divided by the number of instructions in the sample. This simulation speedup metric does not include the overhead of setting the architecture and microarchitecture starting images, as discussed in Section 2, however, state-of-the-art sampled simulation methods use checkpointing to initialize a sampling unit's starting image, for which the overhead only depends on the number of sampling units (to a first-order approximation). In other words, comparing sampling strategies in terms of simulation speedup can be done by simply comparing the number of sampling units (intervals) in the sample versus the entire program execution.

We simulate all sampling units selected by statistical and representative sampling, respectively, and retain the max power consumption of any of these sampling units. We then compare this sampled maximum against the max power consumption observed across the entire benchmark execution — this is done by simulating the complete benchmark execution while keeping track of the max power consumption at the 1M-instruction interval size. The percentage difference between the max power values is called the *error*, which is a smaller-is-better metric: the smaller the error score, the closer the stress pattern identified through sampled simulation reflects the real stress pattern observed across the entire benchmark execution. Figure 2 shows the error in estimating the maximum power consumption. We observe that statistical sampling is less effective in finding stress patterns than representative sampling, i.e., the error can be as high as 60% (and average error of 9.3%) for statistical sampling whereas representative sampling is much more effective. Representative sampling with k-means clustering achieves an average error of 3% (and 14% at most); representative sampling with threshold clustering is even more effective with an average error of 2.3% and a maximum error of at most 11%. The reason for the difference in efficacy between statistical sampling and representative sampling is that representative sampling selects sampling units based on the benchmark execution and structure (through the BBVs that are being collected for finding the distinct phase behaviors), whereas statistical sampling is largely agnostic to any notion of program structure and behavior. In other words, for statistical sampling, the likelihood of hitting upon a stress pattern is inverse proportional to the sampling rate, whereas representative sampling identifies distinct program behavior by looking into the code that is being executed.

The reason why threshold clustering outperforms k-means clustering is that threshold clustering, by construction, bounds the amount of variability within a cluster, whereas

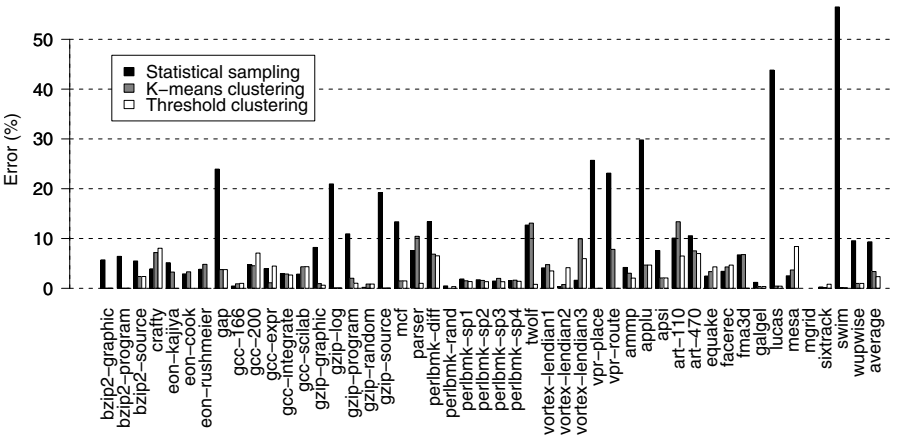


Fig. 2. Error in estimating max power stress patterns

k-means clustering does not. In other words, for a given simulation speedup, i.e., for a given number of clusters, threshold clustering will yield more sparsely populated clusters than k-means clustering; i.e., outliers in the data set will end up in separate clusters in contrast to k-means clustering, which may group those outliers with its closest, albeit relatively far away, cluster.

The end conclusion is that representative sampling with threshold clustering results in a simulation speedup of three orders of magnitude compared to entire benchmark simulation with an error of at most a few percent on average for finding stress patterns in the SPEC CPU2000 benchmarks. And in addition, respresentative sampling with threshold clustering is more effective than representative sampling with k-means clustering and statistical sampling.

4.3 Processor Component Stress Patterns

In the previous section, the focus was on stress patterns for the entire processor. We now look into stress patterns for individual processor components, such as the instruction window, functional units, caches, branch predictor, etc. This, in conjunction with a microprocessor floorplan, could provide valuable information in terms of power density and thermal hotspots [23]. Figures 3 and 4 quantify the error in estimating average and maximum per-component power consumption, respectively. (We assume a $1000\times$ simulation speedup and present average results computed across all benchmarks.) The interesting observation from these graphs is that both statistical and representative sampling are very accurate in estimating average processor component power consumption (the average error is around 1% on average), however, representative sampling is by far more effective in capturing stress patterns. For representative sampling with threshold clustering, the processor component power error for the stress patterns is less than 5%, whereas representative sampling with k-means clustering and statistical sampling lead to an processor component power error of up to 10% and 20%, respectively.

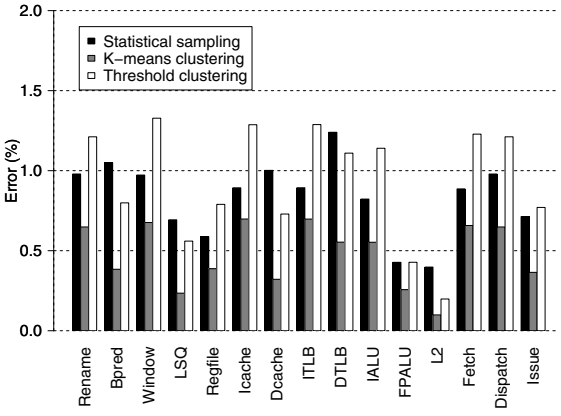


Fig. 3. Error in estimating average power consumption per processor component

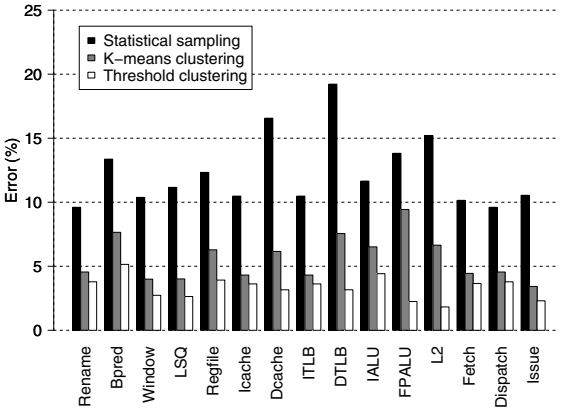


Fig. 4. Error in estimating max power consumption per processor component

4.4 Error Versus Simulation Speedup

The previously reported results assumed a simulation speedup of three orders of magnitude ($1000\times$). We now explore the trade-off between error and simulation speedup in more detail, see Figure 5, which shows two graphs, one for estimating average power consumption (left graph) and another one for estimating max power consumption (right graph) — these graphs show average results across all benchmarks. The vertical and horizontal axes show percentage error and simulation speedup with respect to simulating the entire benchmark, respectively. For computing these graphs, we simulate all sampling units; for the left graph, we then compute the average power consumption across all sampling units, and compare it against the true average power consumption computed by simulating the entire benchmark; for the right graph, we retain the largest power consumption number of any of the sampling units and compare it against the largest power consumption number observed across the entire program execution. For

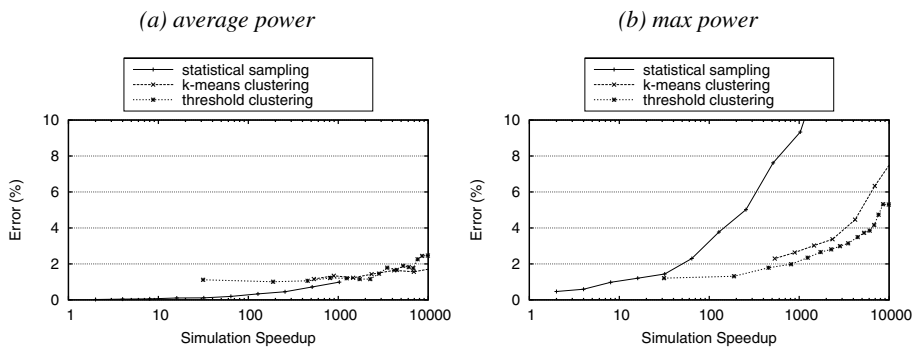


Fig. 5. Statistical sampling versus representative sampling: error as a function of simulation speedup for estimating average power consumption (left graph) and max power (right graph)

statistical sampling, one sampling unit is selected every n intervals; this corresponds to a simulation speedup of a factor n. For representative sampling, we set a maxK parameter or θ threshold for the clustering yielding n clusters or sampling units; this corresponds to a n_{total}/n simulation speedup with n_{total} the number of intervals in the entire program execution.

We observe that statistical sampling is more accurate than representative sampling for estimating average power consumption, see left graph Figure 5. The results in the left graph confirm the earlier findings by Yi et al. [30] who provide a detailed comparison of statistical and representative sampling for estimating average performance: they found that average performance is more accurately estimated through statistical sampling, however, representative sampling has a better speed versus accuracy tradeoff.

However, when it comes to estimating max power consumption, representative sampling is more effective, and threshold clustering is the most effective approach. In particular, representative sampling with threshold clustering finds an interval with a power consumption number around 2% on average of the max power number found through simulation of the entire benchmark at a simulation speedup of three orders of magnitude. For the same simulation speedup, statistical sampling achieves an error of 10% on average. Or, reversely, for an error of 2%, statistical sampling only achieves a simulation speedup around a factor of 40. In other words, representative sampling with threshold clustering is both faster and more effective in capturing max power stress patterns.

4.5 Other Extreme Behaviors

Representative sampling with threshold clustering is effective at finding other flavors of extreme behaviors as well, beyond power related stress patterns. Figure 6 shows four examples, namely max CPI, max L1 D-cache miss rate, max L2 cache miss rate and max branch misprediction rate stress patterns. In all four examples, representative sampling with threshold clustering is the most effective approach; this is especially the case for the CPI and cache miss rate extreme behaviors. These extreme behaviors can provide valuable insight and understanding about problematic program behaviors and patterns.

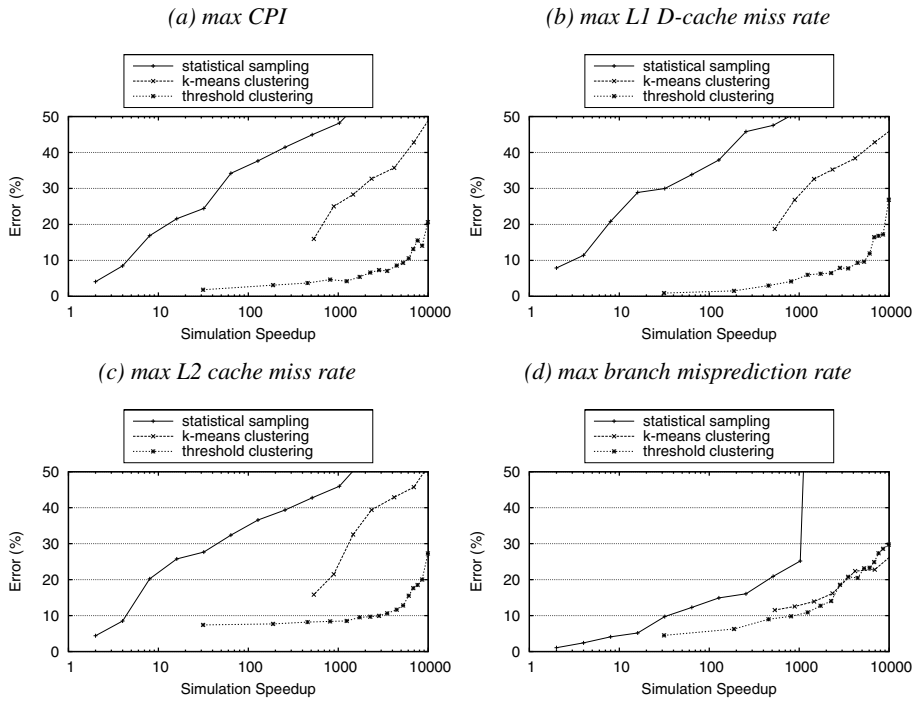


Fig. 6. Finding other flavors of stress patterns: max CPI (top left), max L1 D-cache miss rate (top right), max L2 cache miss rate (bottom left), and max branch misprediction rate (bottom right)

5 Related Work

Stress testing. In VLSI circuit design, statistically generated test vectors are used to stress a circuit by inducing maximum switching activity [4]. At the microarchitectural level, engineers develop hand-crafted synthetic test cases, so called stressmarks, to estimate maximum power consumption of a microprocessor. This is common practice in industry, see for example [9,10,27]. Recent work by Joshi et al. [17] proposes a framework for automatically developing stressmarks by exploring the workload space using an abstract workload model.

Power phase characterization. A lot of work has been done on characterizing time-varying program behavior, and different authors have been proposing different ways for doing so, such as code working sets [6], BBVs [22], procedure calls [13], and performance data [7].

Isci and Martonosi [14] propose a methodology for tracking dynamic power phase behavior in real-life applications using a real hardware setup. They measure total processor power consumption data using a digital multimeter and simultaneously collect raw performance counter data. They then use the performance counter data to estimate processor component power consumption numbers, which they subsequently use to

identify power phase behavior at runtime using threshold clustering. Whereas the goal of the work by Isci and Martonosi is on tracking power consumption and power phase behavior at runtime, the focus of our work is on finding stress patterns to guide processor design under extreme workload behavior, which is a related but different problem.

In their follow-on work, Isci and Martonosi [15] compare clustering based on BBVs versus processor component power numbers, and found both approaches to be effective, but processor component power numbers to be more accurate for tracking power phase behavior. The downside of processor component power numbers though is that it requires that the entire benchmark be measured in terms of its power behavior, which may be costly in terms of equipment (in case of a real hardware setup) or which may be too time-consuming (in case of a simulation setup). In addition, processor component power numbers are specific to one particular microprocessor implementation. A BBV profile is both inexpensive and fast to measure through software instrumentation, and, in addition, is microarchitecture-independent, i.e., can be used across microarchitectures. Since our goal is to find stress patterns to be used during the design of a processor, we advocate the BBV approach because of its microarchitecture-independence, its low cost and its fast computation.

6 Conclusion and Future Work

Power consumption has emerged as a key design concern over the entire range of computing devices, from embedded systems up to large-scale data centers and supercomputers. Understanding the power characteristics of workloads and their interaction with the architecture however, is not trivial and requires an appropriate benchmarking methodology. Researchers and engineers currently use a range of workloads for gaining insight into the power characteristics of processor architectures. On the one side, typical workloads such as SPEC CPU and other commercial workloads are used to assess average power consumption. On the other side, hand-crafted stressmarks are being used to understand worst-case behavior in terms of a processor's max power consumption. This paper closed the gap between these two ends of the power benchmarking spectrum by finding stress patterns in typical microprocessor workloads.

In this paper, we advocated and studied sampled simulation as a means of finding these stress patterns efficiently. Although sampled simulation is a well studied and mature research area, the objective in this paper is completely different. While the goal of sampled simulation traditionally has been on estimating average performance, the problem addressed in this paper is on estimating worst-case performance rather than average performance, i.e., the goal is to find stress patterns in typical workloads without having to simulate the complete benchmark execution. We found that although statistical sampling is more effective than representative sampling for estimating average behavior, representative sampling is substantially more effective than statistical sampling when it comes to capturing extreme behavior. In addition, we found that threshold clustering is substantially more effective than k-means clustering for finding stress patterns (which is a frequently used clustering technique for representative sampling). Our experimental results using the SPEC CPU2000 benchmarks demonstrate that stress patterns at a million-instruction granularity can be found with an error of a few percent on average at a simulation speedup of three orders of magnitude.

We believe that this work could lead to a new line of research towards finding stress patterns in microprocessor workloads. Sampled simulation, which was traditionally used for estimating average behavior, may benefit from specific enhancements towards stress pattern identification. One focus of future research may be to improve the computational requirements of the clustering algorithm in representative sampling so that larger data sets and thus smaller granularity stress patterns may become feasible in practice. One example of a stress pattern that requires a small granularity is a dI/dt stress pattern: stress patterns with large power swings over short periods of time are of interest for studying the dI/dt problem [16] as the associated current swings may lead to ripples on the voltage supply lines, which may introduce timing errors and/or cause circuits to fail. Existing clustering algorithms however are too time-consuming when applied to a large data set.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. Lieven Eeckhout is a postdoctoral fellow with the Fund for Scientific Research in Flanders (Belgium) (FWO-Vlaanderen). Additional support is provided by the FWO projects G.0160.02 and G.0255.08.

References

1. Brooks, D., Martonosi, M.: Dynamic thermal management for high-performance microprocessors. In: HPCA, pp. 171–182 (January 2001)
2. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: A framework for architectural-level power analysis and optimizations. In: ISCA, pp. 83–94 (June 2000)
3. Burger, D.C., Austin, T.M.: The SimpleScalar Tool Set. Computer Architecture News (1997), `http://www.simplescalar.com`
4. Chou, T., Roy, K.: Accurate power estimation of CMOS sequential circuits. IEEE Transaction on VLSI Systems 4(3), 369–380 (1996)
5. Conte, T.M., Hirsch, M.A., Menezes, K.N.: Reducing state loss for effective trace sampling of superscalar processors. In: ICCD, pp. 468–477 (October 1996)
6. Dhodapkar, A., Smith, J.E.: Managing multi-configuration hardware via dynamic working set analysis. In: ISCA, pp. 233–244 (May 2002)
7. Duesterwald, E., Cascaval, C., Dwarkadas, S.: Characterizing and predicting program behavior and its variability. In: PACT, pp. 220–231 (October 2003)
8. Eeckhout, L., Luo, Y., De Bosschere, K., John, L.K.: BLRL: Accurate and efficient warmup for sampled processor simulation. The Computer Journal 48(4), 451–459 (2005)
9. Felter, W., Keller, T.: Power measurement on the Apple Power Mac G5. Technical Report RC23276, IBM (2004)
10. Gowan, M.K., Biro, L.L., Jackson, D.B.: Power considerations in the design of the Alpha 21264 microprocessor. In: Proceedings of the 35th Design Automation Conference (DAC), pp. 726–731 (June 1998)
11. Gunther, S.H., Binns, F., Carmean, D.M., Hall, J.C.: Managing the impact of increasing microprocessor power consumption. Intel. Journal of Technology 5(1) (February 2001)
12. Haskins Jr., J.W., Skadron, K.: Accelerated warmup for sampled microarchitecture simulation. ACM Transactions on Architecture and Code Optimization (TACO) 2(1), 78–108 (2005)

13. Huang, M., Renau, J., Torrellas, J.: Positional adaptation of processors: Application to energy reduction. In: ISCA, pp. 157–168 (June 2003)
14. Isci, C., Martonosi, M.: Runtime power monitoring in high-end processors: Methodology and empirical data. In: MICRO, pp. 93–104 (December 2003)
15. Isci, C., Martonosi, M.: Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In: HPCA, pp. 122–133 (February 2006)
16. Joseph, R., Brooks, D., Martonosi, M.: Control techniques to eliminate voltage emergencies in high performance processors. In: HPCA, pp. 79–90 (February 2003)
17. Joshi, A.M., Eeckhout, L., John, L.K., Isen, C.: Performance cloning: A technique for disseminating proprietary applications as benchmarks. In: HPCA, pp. 229–239 (February 2008)
18. Kanter, D.: EEMBC energizes benchmarking. Microprocessor Report (July 2006)
19. Kluyskens, S., Eeckhout, L.: Branch history matching: Branch predictor warmup for sampled simulation. In: De Bosschere, K., Kaeli, D., Stenström, P., Whalley, D., Ungerer, T. (eds.) HiPEAC 2007. LNCS, vol. 4367, pp. 153–167. Springer, Heidelberg (2007)
20. Laha, S., Patel, J.H., Iyer, R.K.: Accurate low-cost methods for performance evaluation of cache memory systems. IEEE Transactions on Computers 37(11), 1325–1336 (1988)
21. Lau, J., Sampson, J., Perelman, E., Hamerly, G., Calder, B.: The strong correlation between code signatures and performance. In: ISPASS, pp. 236–247 (March 2005)
22. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: ASPLOS, pp. 45–57 (October 2002)
23. Skadron, K., Stan, M.R., Huang, W., Velusamy, S., Sankaranarayanan, K., Tarjan, D.: Temperature-aware microarchitecture. In: ISCA, pp. 2–13 (June 2003)
24. SPEC. Specpower_ssj2008, `http://www.spec.org/power_ssj2008/`
25. Tiwari, V., Singh, D., Rajgopal, S., Mehta, G., Patel, R., Baez, F.: Reducing power in high-performance microprocessors. In: DAC, pp. 732–737 (June 1998)
26. Van Biesbrouck, M., Eeckhout, L., Calder, B.: Efficient sampling startup for sampled processor simulation. In: Conte, T., Navarro, N., Hwu, W.-m.W., Valero, M., Ungerer, T. (eds.) HiPEAC 2005. LNCS, vol. 3793, pp. 47–67. Springer, Heidelberg (2005)
27. Vishmanath, R., Wakharkar, V., Watwe, A., Lebonheur, V.: Thermal performance challenges from silicon to systems. Intel. Technology Journal 4(3) (August 2000)
28. Wenisch, T.F., Wunderlich, R.E., Falsafi, B., Hoe, J.C.: Simulation sampling with live-points. In: ISPASS, pp. 2–12 (March 2006)
29. Wunderlich, R.E., Wenisch, T.F., Falsafi, B., Hoe, J.C.: SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In: ISCA, pp. 84–95 (June 2003)
30. Yi, J.J., Kodakara, S.V., Sendag, R., Lilja, D.J., Hawkins, D.M.: Characterizing and comparing prevailing simulation techniques. In: HPCA, pp. 266–277 (February 2005)

Deriving Efficient Data Movement from Decoupled Access/Execute Specifications

Lee W. Howes[1], Anton Lokhmotov[1], Alastair F. Donaldson[2], and Paul H.J. Kelly[1]

[1] Department of Computing, Imperial College London
180 Queen's Gate, London, SW7 2AZ, UK
[2] Codeplay Software, 45 York Place, Edinburgh, EH1 3HP, UK

Abstract. On multi-core architectures with software-managed memories, effectively orchestrating data movement is essential to performance, but is tedious and error-prone. In this paper we show that when the programmer can explicitly specify both the memory access pattern and the execution schedule of a computation kernel, the compiler or run-time system can derive efficient data movement, even if analysis of kernel code is difficult or impossible. We have developed a framework of C++ classes for decoupled Access/Execute specifications, allowing for automatic communication optimisations such as software pipelining and data reuse. We demonstrate the ease and efficiency of programming the Cell Broadband Engine architecture using these classes by implementing a set of benchmarks, which exhibit data reuse and non-affine access functions, and by comparing these implementations against alternative implementations, which use hand-written DMA transfers and software-based caching.

1 Introduction

Architectures with software-managed memories can achieve higher performance and power efficiency than traditional architectures with hardware-managed memories (*e.g.* caches), but place additional burden on the programmer. For a traditional architecture, the programmer typically designs a computation kernel and specifies the order in which the kernel traverses the iteration space. To off-load the kernel to a co-processor equipped with local memory, the programmer must additionally manage data movement, to ensure that data is smoothly streamed to and from local memory.

This additional step sounds easier than it actually is. The performance-conscious programmer needs to consider issues such as the optimal data transfer sizes, alignment constraints, exploiting data reuse, *etc.* Moreover, when the working set of a processor is too large to fit in its local memory, the programmer has to use low-level optimisation techniques such as double buffering to overlap computation and communication. Unfortunately, this harms code portability and maintainability.

In this paper, we introduce the decoupled Access/Execute (Æcute—pronounced "acute") programming model, which allows the programmer to express explicitly both the memory access pattern and the execution schedule of a computation kernel, similar to programming traditional architectures. We show that in many cases the compiler or run-time system can derive efficient data movement even if analysis of kernel code is

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 168–182, 2009.

difficult or impossible, thus removing from the programmer the additional complexity of managing data movement.

In the remainder of this paper we argue that decoupling access and execute is natural when programming architectures with software-managed memories (§2) and introduce decoupled Access/Execute specifications (§3). We discuss the prototype Æcute framework (§4) and use examples adapted from linear algebra and signal processing (§3.1 and §5) to show the ease of programming using the specifications. We present experimental results for our examples (§6) obtained on a Cell Broadband Engine (BE) processor and compare them against alternative implementations, which use hand-written DMA transfers and software-based caching. We review related work (§7) and conclude with an outline of future work (§8).

2 Background

Since the 1980s, microprocessor designers have worked hard to preserve the *illusion* of fast memory by providing hardware-managed caches. Sadly, increasing the number of transistors dedicated to caches has been found to achieve diminishing effects on performance. Moreover, optimising software for the memory hierarchy has become the principal activity of performance-conscious programmers and compiler writers, who "spend much of their time *reverse-engineering and defeating* the sophisticated mechanisms that automatically bring data on to and off the chip" [1].

Given this unsatisfactory situation, designers have turned their attention to software-managed memory hierarchies, where data is copied between memories under explicit software control. Examples include the Cell BE architecture from Sony/Toshiba/IBM [1], the CSX SIMD array architecture from ClearSpeed [2], and massively parallel architectures from NVIDIA and ATI (still habitually called graphics processing units, GPUs).

Local memory is typically cheap to access (*e.g.* 6 cycles on Cell), and thus is akin to an extended register file. One some architectures (*e.g.* on Cell and CSX), processing elements can only access local memory, and need to invoke expensive data transfer mechanisms to access remote memory. On other architectures (*e.g.* on GPUs), exploiting local memory is not obligatory but is essential to performance.

Efficient programs are naturally separated into two parts: remote memory access to copy operands in and to copy results out (often asynchronously), and execution in local memory to produce the results. The access and execute parts can be thought of as two concurrent instruction streams. For example, on Cell the execute part runs on an SPE, while the access part is serviced by its DMA engine (Memory Flow Controller).

The separation is reminiscent of decoupled access/execute architectures [3], which run (conceptually or physically) separate access and execute instruction streams. Another point of reference is data-prefetching in virtual shared memory, a shared memory abstraction for computers with physically distributed memories [4]. A program runs on two processors: the access processor prefetches remote data into local memory by performing a partial evaluation of the program; the execute processor performs the full evaluation. The scout-threads in Sun's upcoming Rock processor [5] manifest the same idea, by reading the instruction stream ahead during a memory access stall.

Ideally, to hide the memory latency the access stream runs well in advance of the execute stream. Occasionally the streams need to synchronise, for example, when the execute stream computes an address required by the access stream. Topham *et al.* [6] describe special compiler techniques to minimise the frequence of such *loss of decoupling* events.

Decoupled architectures use either a single original program or two programs derived (manually or automatically) from the original program. We observe, however, that deriving access and execute instruction streams from programs written in mainstream programming languages such as C/C++ is hard, in particular, because of the difficulty of dependence analysis in the presence of aliasing.

3 Decoupled Access/Execute Specifications

We propose a declarative programming model that allows the programmer to annotate a computation kernel with both the execute (§3.2) and access (§3.3) metadata.

3.1 Motivating Example: The Closest-to-Mean Image Filter

The *Closest-to-Mean* (CTM) filter [7] is an effective mechanism for reducing noise in near Gaussian environments, preserving edges more effectively than linear filters whilst offering better performance than computationally expensive median-based filters. For a sample set of vectors V with distance metric δ, the output for the CTM filter is given by the following formula:

$$CTM(V) = \arg\min_{x \in V} \delta(x, \overline{x}),$$

where $\overline{x}$ denotes the sample average value, and $\arg\min_{x \in V}(\textit{expr})$ denotes a value of x that minimises *expr*.

The CTM filter can be applied to a digital $W \times H$ image by mapping each pixel to a CTM value for a $(2K+1) \times (2K+1)$ square sample of neighbouring pixels (for some $K > 0$). Fig. 1 shows a CTM filter implementation in our prototype C++ framework.[1] The class method `kernel` closely resembles the filter's original kernel code, except that accesses to arrays have been replaced with uses of Æcute access descriptors (§4.1).

3.2 Execute Metadata

Definition 1. *Execute metadata for a kernel is a tuple $E = (I, R, P)$, where:*

- $I \subset \mathbb{Z}^n$ *is a finite, n-dimensional* iteration space*, for some $n > 0$;*
- $R \subseteq I \times I$*, is a precedence relation such that $(i_1, i_2) \in R$ iff iteration i_1 must be executed before iteration i_2.*
- P *is a* partition *of I into a set of non-empty, disjont iteration subspaces.*

[1] Note that in all our examples we have compacted construction of member fields into their declarations, to save space.

```
class CTMFilter : public StreamKernel {
  Neighbourhood2D_Read inputPointSet(iterationSpace, input, K);
  Point2D_Write outputPointSet(iterationSpace, output);

  CTMFilter( IterationSpace2D &iterationSpace,
    int K, Array2D &input, Array2D &output ) {...}
  ...
  void kernel( const IterationSpace2D::element_iterator &eit ) {
    // compute mean
    rgb mean( 0.0f, 0.0f, 0.0f );
    for(int w = -K; w <= K; ++w) {
      for(int z = -K; z <= K; ++z) {
        mean += inputPointSet(eit, w, z); // input[x+w][y+z]
      }
    }
    mean /= (2*K + 1) * (2*K + 1);
    // compute closest to mean
    rgb closest = inputPointSet(eit, 0, 0); // input[x][y]
    for(int w = -K; w <= K; ++w) {
      for(int z = -K; z <= K; ++z) {
        rgb curr = inputPointSet(eit, w, z); // input[x+w][y+z]
        if( dist(curr, mean) < dist(closest, mean) )
          closest = curr;
      }
    }
    outputPointSet(eit) = closest; // output[x][y]
  }
}
```

Fig. 1. Æcute implementation code for the CTM filter

```
const int K = 2; // 5x5 filter

// 2D iteration space is equivalent to a doubly nested loop:
// parallel for (int x = K; x < W-K; ++x)
//   parallel for(int y = K; y < H-K; ++y)
IterationSpace2D iterationSpace( K, W-K, K, H-K );

// 2D array descriptors
Array2D < rgb >  inputArray( W, H,  &input[0][0] );
Array2D < rgb > outputArray( W, H, &output[0][0] );

// Filter class instantiation
CTMFilter filter( iterationSpace, K, inputArray, outputArray );

// Filter invocation
filter.execute();
```

Fig. 2. Æcute setup and invocation code for the CTM filter

The precedence relationship R specifies constraints on the execution schedule: if iterations i_1 and i_2 are in the relationship, i_1 must be executed before i_2; otherwise, i_1 and i_2 can be executed in any order.

The partition P indicates sets of iterations that it is sensible to execute on the same processing element (*e.g.* a set of iterations that exhibit data reuse). In this paper, we assume that the working set of each $p \in P$ fits into local memory, assuming a set number of buffers (*e.g.* two for double buffering); the programmer either partitions the iteration space manually or opts to use a simple automatic partitioning method which computes the maximum iteration subspace size based on this constraint.

In the CTM filter example, the iteration space is a two-dimensional rectangle congruous with the image dimensions; if the input and output images are disjoint, the execution schedule can be unconstrained, maximising parallelism; and the partition can be tiling into rectangular $w \times h$ tiles, maximising locality:[2]

- $I = \big\{(x, y) : K \leq x < W - K, K \leq y < H - K\big\}$
- $R = \emptyset$
- $P = \big\{\{(x, y) \in I : w(i-1) \leq x - K < wi, h(j-1) \leq y - K < hj\} :$
  $1 \leq i < (W - 2K)/w, 1 \leq j < (H - 2K)/h\big\}$

3.3 Access Metadata

Let M be a set of memory locations.

Definition 2. *Access metadata for a kernel is a tuple $A = (M_r, M_w)$, where:*

- $M_r : I \to \mathcal{P}(M)$ *specifies the set of memory locations $M_r(i)$ that may be* read *on iteration $i \in I$;*
- $M_w : I \to \mathcal{P}(M)$ *specifies the set of memory locations $M_w(i)$ that may be* written *on iteration $i \in I$.*

Often, the set of memory locations accessed on a given iteration is a function of the *iteration vector* (in which case we say that the set is *indexed* by the iteration vector); the set can also include locations that are independent of the iteration vector such as scalars.

In the CTM filter example, the input and output memory locations are indexed:

- $M_r = \big\{\mathrm{input}[x][y] : (x, y) \in I\big\}$;
- $M_w = \big\{\mathrm{output}[x + w][y + z] : (x, y) \in I, -K \leq w, z \leq K\big\}$.

3.4 Æcute Specifications

Definition 3. *An Æcute specification for a kernel is a tuple $S = (A, E)$, where A and E are its access and execute metadata.*

Access metadata 'knows' about memory locations that may be accessed on each iteration, while execute metadata 'knows' about iteration subspaces that are to be executed.

[2] We assume, for simplicity, that the iteration space contains a whole number of tiles.

Given an iteration subspace $p \in P$ and access metadata, we can (over) approximate the set of memory locations that the subspace may read and write: $M_r(p) = \{M_r(i) : i \in P\} \in \mathcal{P}(L)$ and $M_w(p) = \{M_w(i) : i \in P\} \in \mathcal{P}(L)$. Combining execute and access metadata in the form of Æcute specifications enables powerful optimisations such as software pipelining and exploiting data reuse.

In the CTM filter example, Æcute specifications can be used to trigger data prefetching of image rows into local memory, to ensure that the data is delivered in time for processing.

4 Æcute C++ Framework

We have developed a prototype framework to support the Æcute concept, consisting of a set of C++ descriptor classes (§4.1) and a run-time system (§4.2), which compile for the Cell BE architecture.

4.1 The Æcute C++ Classes

The formal iteration space I is specified via an instance of an `IterationSpace` class, which records the number of dimensions and size of each dimension, as in Fig. 2. Practically useful timestamp functions (T) are available in our prototype via the definition of serialised dimensions on iteration spaces, *e.g.* using the `COLUMN_SERIAL` directive. Partitioning of the iteration space is performed in the current prototype with a call to the `setBlockSize` function, which is parameterised with the size of a partition in each dimension of the iteration space.

A kernel class contains a main kernel method parameterised by an iterator to be executed on each point in the iteration space (*e.g.* see Fig. 1). The iterator is used to access indexed memory locations.

The memory mappings M_r and M_w are defined by *access descriptor* classes. An access descriptor object is created for each input or output associated with a kernel, and is invoked from the kernel code, parameterised by an iterator, to gain access to data. The prototype implementation supports the following access descriptor classes. For each member of the iteration space:

- `Point`: returns a single element of the data structure.
- `Neighbourhood`: returns a set of memory locations within a given radius of a primary address.
- `Buffer`: returns a set of points with per point addressing into the data structure based on a combination of the primary address and buffer offset.

In each case the primary address is computed from the iteration space coordinates provided by the Æcute iterator. To these coordinates we may apply a conversion function. In the examples of §5 we see `Project`, `ReAddress`, `Identity` and `BitRev`. `Project` is an affine scaling function. `ReAddress` is a proxy for applying separate conversions to each dimension: in our examples, the identity function `Identity` and a custom bit-reversal function, `BitRev`. The prototype framework can be extended with custom conversion functions for specific applications.

4.2 The Æcute Run-Time System

The Æcute run-time system comprises two components: a PPE run-time and an SPE run-time, an instance of which runs on each active SPE.

The PPE run-time spawns an SPE run-time process on each available SPE. Based on an iteration space partitioning specified by the programmer, or via a partitioning obtained automatically at run-time by querying access metadata, the PPE run-time generates a list of partition identifiers. Partition identifiers are farmed out to the SPEs, which are responsible for executing the kernel iterations associated with each partition. Once all partitions have been assigned, the PPE run-time waits for completion reports from all SPEs before returning control to the main program.

On initialisation all access descriptors in the SPE instance create a series of buffers based on the maximum partition size. At least one buffer will be present in each input and output descriptor, and possibly more if the configuration specifies this.

An instance of the SPE run-time repeatedly receives a list of partition identifiers from the PPE. The SPE instance takes each partition identifier in turn and converts that into a set of iterations. The conversion is possible because the SPE code is constructed using the same iteration space data as the PPE code. This information is partially static, and partially based on parameters passed on construction from the PPE side.

The partition information is passed to the access descriptors assigned to the kernel, which select available data buffers and construct appropriate DMA operations to copy data in. When no buffer is available, a blocking call to wait on DMA writes is initiated to allow buffers to be cleared and reused. The kernel checks that the data it needs for a given partition identifier is available by querying the access objects, and will block on the DMA reads if it is not. On completion of computation, partition completion information is passed to the access objects which will perform DMA write backs and free buffers as appropriate.

Double or triple buffering naturally occurs through this system, as a fixed buffer set is automatically managed to ensure that data is always available, without additional programmer intervention. This multiple buffering enables dynamic software pipelining of the execution to improve the efficiency of memory access. In addition, the run-time system will maintain buffers without reloading, or without writing back early, if it detects that it already had the appropriate data resident in an appropriate buffer.

5 Further Examples

5.1 Matrix-Vector Multiply

A matrix-vector multiply $y = Ax$ can be implemented as a two-dimensional iteration space of the dimensions of matrix A. Vectors x and y are one-dimensional, so we *project* the iteration space to obtain the vector indices.

Æcute specification $S = ((M_r, M_w), (I, T, P))$:

- $I = \{(i, j) : 0 \leq i < H, 0 \leq j < W\}$
- $R = \{((i, j), (i, k)) : 0 \leq i < H, 0 \leq j < k < W\}$

- $M_r(i,j) = \{A[i][j], x[j]\}$
- $M_w(i,j) = \{y[i]\}$
- $P = \big\{\{(i,j) \in I : h(k-1) \leq i < hk, w(l-1) \leq j < wl, \} :$
  $1 \leq k < H/h, 1 \leq l < W/w\big\}$

(As before, we tile the iteration space, assuming local memory can hold the working set for a rectangular tile of $h \times w$ iterations.)

The precedence relation indicates that the loop indexed by i is parallel and the loop indexed by j is serial. This serialisation removes the requirement for PPE-side accumulation of partial results. If the += operator could be guaranteed to be associative then the j loop could also be specified as parallel, by setting $R = \emptyset$.

Æcute code. The kernel operates over the input matrix and vector and the output vector. Note that we specify that the column dimension is serial, which preserves the order of multiply-accumulate operations.

```
IterationSpace2D iterationSpace(W, H, COLUMN_SERIAL);
Array2D < float >  inMatrix(H, W, pInMatrix);
Array1D < float >  inVector(W, pInVector);
Array1D < float > outVector(H, pOutVector);
MatrixVectorMul matvec(iterationSpace, inMatrix, inVector, outVector);
// Matrix-vector multiply invocation
matvec.execute();
```

The `MatrixVectorMul` kernel class is roughly as follows:

```
class MatrixVectorMul : public StreamKernel {
  Point2D_Read inputMatrix( iterationSpace, inMatrix);
  Point2D_Read < Project2D1D< 1, 0 >  >
    inputVector( iterationSpace, inVector );
  Point2D_Write < Project2D1D< 0,1 >  >
    outputVector( iterationSpace, outVector );

  MatrixVectorMul( IterationSpace 2D iterationSpace,
    Array2D inMatrix, Array1D inVector, Array1D outVector ){...}

  void kernel( const IterationSpace2D::element_iterator &eit ) {
    outputVector( eit ) += inputVector( eit ) * inputMatrix( eit );
  }
};
```

where `Project2D1D` projects a 2D-space point onto a 1D-space point. For example, `Project2D1D<0,1>` projects (i,j) onto j.

5.2 Bit-Reversal

Many radix-2 FFT algorithms start or end their processing with data permuted in bit-reversed order. The reordering is typically done by a special subroutine, called

bit-reversed data copy (often abbreviated, if inaccurately, to *bit-reversal*). We assume that the subroutine reads an array $x[\,]$ of $N = 2^n$ elements and writes these elements into an array $y[\,]$ of N elements, such that x and y do not overlap, in bit-reversed order. That is, an element of the source array at the index written in binary as $b_0 \ldots b_{n-1}$, is copied to the target array at the index with reversed digits $b_{n-1} \ldots b_0$. The function $\sigma_n(i)$ reversing bits of index i having n bits can be implemented as [8]:

```
unsigned int reverse_bits(unsigned int n, unsigned int i) {
    i = (i & 0x55555555) << 1 | (i >> 1) & 0x55555555;
    i = (i & 0x33333333) << 2 | (i >> 2) & 0x33333333;
    i = (i & 0x0f0f0f0f) << 4 | (i >> 4) & 0x0f0f0f0f;
    i = (i<<24) | ((i & 0xff00)<<8) | ((i>>8) & 0xff00) | (i>>24);

    return (i >> (32 - n));
}
```

Few programmers will recognise that this sequence of bit-wise operations and shifts implies that $y[\,]$ will contain a permutation of $x[\,]$ and hence assignments can be performed in any order. One cannot expect that a compiler will recognise this either.

In addition to obscuring parallelism, bit-reversed indexing is unfriendly to hardware-managed caches: starting from a certain array size $N = 2^n$, each access to $y[\,]$ results in a cache miss. To avoid cache associativity problems inherent in bit-reversals of large arrays, the best approach, used by Carter and Gatlin in the so-called Cache Optimal BitReverse Algorithm (COBRA) [9], introduces a cache-resident buffer.

If the buffer holds B^2 elements, the iteration space is partitioned into N/B^2 independent subspaces. For each subspace, B source blocks of B elements each are copied into the buffer, permuted *in place*, and then copied out from the buffer into B target blocks of B elements each.

The permute kernel of COBRA can be off-loaded to a co-processor having local memory. The challenge is to implement the copy in and copy out loops, where the copy out loop uses a non-affine access function $\sigma_n(i)$.

Somewhat surprisingly, implementing data movement code can take longer than implementing the kernel proper (according to the experience of one of the authors). Again, a desired alternative is to derive data movement from Æcute specifications.

Æcute specification $S = \big((M_r, M_w), (I, R, P)\big)$:

- $I = \{t : 0 \le t < N/B^2\}$
- $R = \emptyset$;
- $P = \big\{\{t\} : t \in I\big\}$
- $M_r(t) = \{x[u.t.v] : t \in I, 0 \le u < B, 0 \le v < B\}$
- $M_w(t) = \{y[u.\sigma_n(t).v] : t \in I, 0 \le u < B, 0 \le v < B\}$.

The precedence function indicates that the one-dimensional iteration space is unordered. In this case each partition is a single element of the iteration space, because the blocks are disjoint and fairly large. In the Æcute code below we see that the programmer can manually set the partition size.

Ækute code As a result of the $B \times B$ blocking, it is natural to think of the input and output arrays of N elements as two-dimensional, having N/B rows of B elements each.

```
IterationSpace1D iterationSpace(N/(B*B));
Array2D <float> inputData(B, N/B,  pInputData);
Array2D <float> outputData(B, N/B, pOutputData);
BitReversal bitrev(iterationSpace, inputData, outputData);
bitrev.iterationSpace.setBlockSize( 1 );
// Bit-reversal invocation
bitrev.execute();
```

We iterate over independent subspaces $t \in I$, copying rows numbered as $u.t, 0 \leq u < B$, into the local buffer, applying the kernel, and copying rows numbered as $u.\sigma_n(t), 0 \leq u < B$, from the local buffer.

```
class BitReversal : public StreamKernel< BitReversal > {
    Buffer2D_Read
        input(iterationSpace, inputData, B);
    Buffer2D_Write < ReAddress2D< Identity, BitRev > >
        output(iterationSpace, outputData, B);

    BitReversal( IterationSpace 2D iterationSpace,
        Array2D input, Array2D output ) {...}

    // Do in place permutation
    void kernel(const IterationSpace2D::element_iterator &eit){
        ...
    }
};
```

`ReAddress` takes the (i, j) coordinate formed from the iteration space point and the buffer coordinates and applies the specified pair of functions to i and j respectively. `BitRev` reverses bits of the j value to correctly address the destination for the row by calling the `reverse_bits` function shown earlier.

6 Experimental Evaluation

We use a 3.2GHz Cell processor on a Sony PlayStation 3 console, running Fedora Linux (2.6.23.17-88.fc7), with IBM Cell SDK 2.1. We compiled the benchmark programs using the highest optimisation settings, and executed them on all six SPEs that are available to the programmer on a PlayStation 3.

6.1 Implementation

We evaluate our prototype Ækute framework against alternative implementations, which use hand-written DMA transfers and a software-based SPE cache. The cache allows remote data to be accessed in a familiar way, so that code can be quickly ported to run on an SPE. In our experiments, we use the standard cache implementation provided with SDK 2.1 [10]. We use a 4-way set associative cache with default "write-back" write policy and "round-robin" replacement policy, and vary the number of cache sets and line size on an application-specific basis.

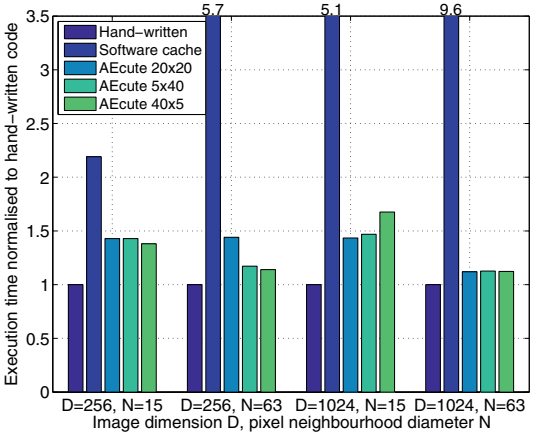


Fig. 3. Closest-to-mean filter

The kernel code is essentially the same, with minor changes to support the use of Æcute framework classes and software cache functions.

6.2 Benchmark Details

We evaluate the benchmarks described in §3 and §5.

Closest-to-mean filter (§3.1). Fig. 3 shows execution time normalised to code with hand-written DMA transfers. We consider two neighbourhood diameters N: 15 and 63, and two image sizes $D \times D$ where D is: 256 and 1024. These represent increasing computation workload. We also consider three different iteration space tile sizes: 20×20 (default square size, which is calculated automatically under the constraint that the tile footprint must fit into local memory); 5×40; and 40×5.

For $D = 256$ and $N = 15$, the best Æcute code performs within 40% of hand-written code; for $N = 63$, within 15%: the increased workload amortises the overhead of interpreting Æcute specifications. In contrast, the overhead of using the software cache grows with increasing neighbourhood size (which perhaps can be remedied by tuning the cache parameters). For $D = 1024$ and $N = 63$, the overhead drops to 12%.

We observe that no tile size was universally best. Given the simplicity of varying tile sizes, the best tile size could be found by iterative search. In contrast, it is usually more difficult to adapt code with hard-coded tile sizes.

Blocked DMA transfers, which are supported naturally by the partitioning and automated buffering in the Æcute system, and implemented in the hand written code, improve the efficiency of memory traffic and enable both hand-written and Æcute code perform far better than code using the software cache.

Matrix-vector multiply (§5.1). For this example we hand-vectorised the entire block computation for efficiency. The hand written and software cache based code are similarly vectorised for fair comparison. While the Æcute model looks promising for

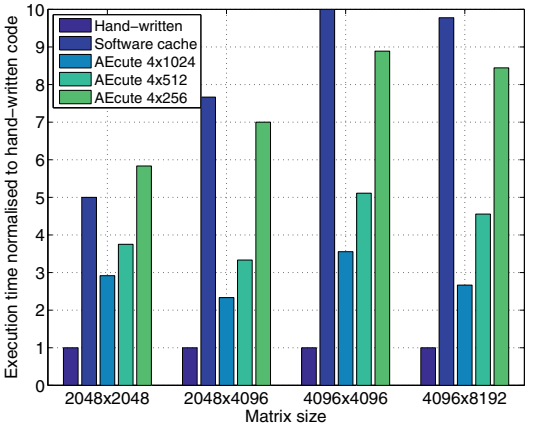


Fig. 4. Matrix-vector multiply normalised to execution time of hand written code

automatic vectorisation, it is important that the programmer retains full control over kernel optimisations should automatic optimisations fail.

Fig. 4 shows normalised execution time for various matrix sizes. The best tile size is 2–3 slower than hand-written code, but considerably faster than the software cache implementation. The run-time overhead associated with the Æcute framework is significant for this example due to the low arithmetic density of the matrix-vector multiply operation. The hand-written implementation requires less SPE-PPE communication: the SPEs are able to compute results entirely independently.

Bit-reversal (§5.2). Fig. 5 plots execution time in milliseconds against $n = \log_2 N$, the bitwidth of the array index. We see smooth scaling of performance with the size of the dataset. In addition, the performance of the Æcute implementation tracks that of the hand-written implementation with a near-constant scaling. In this case, while remote memory accesses are inherently non-contiguous due to bit-reversed indexing in the algorithm, the system can construct efficient DMA list transfers from Æcute specifications.

7 Related Work

Recent work by Solar-Lezama *et al.* on sketching [11] aims to automate the optimisation of simple computation kernels. Where the Æcute model defines iteration spaces and memory access patterns declaratively to localise memory access, sketching supports a rough definition of an optimised implementation and attempts to search for a series of transformations to convert one to the other.

Saltz *et al.* [12] propose run-time parallelisation of loop nests that defy compile-time dependence analysis. At run-time, *inspector* procedures identify parallel wavefronts of loop iterations, which *executor* procedures then distribute among processors. In contrast, our approach relies on the programmer to supply information that the compiler may fail to extract from the program.

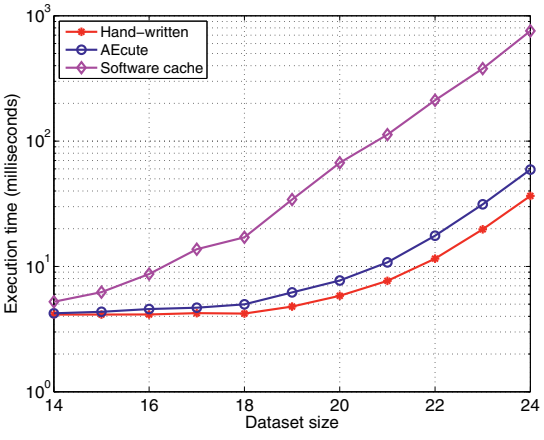


Fig. 5. Bit-reversal

The emergence of architectures having software-managed memories (in particular, of Cell) has spurred the development of high-level programming abstractions, addressing the issue of copying data between distributed memories.

Sequoia. The Sequoia language [13] from Stanford University abstracts parallelism and communication by introducing *tasks*: side-effect free methods using the call-by-value-result (CBVR) argument passing mechanism. The abstract machine model of Sequoia is a tree of (physical or virtual) memory modules. Each task runs at a single node of the tree and can directly access memory only at this node. Tasks can spawn subtasks on the same node or child nodes. Upon calling a subtask, input data from the caller's address space is copied into the callee's address space, output data is computed and then copied out into the caller's address space on return.

CellSs. CellSs [14] is a programming model for the Cell architecture from Barcelona Supercomputing Centre. Similar to Sequoia, CellSs annotations to C programs specify a task for execution on the SPEs and its arguments.

Sieve C++. In Sieve C++ [15][16], a C++ extension from Codeplay Software, the programmer can place a code fragment inside a *sieve scope*—a new lexical scope prefixed with the **`sieve`** keyword—thereby instructing the compiler to *delay* writes to memory locations defined outside of the scope (global memory), and apply them *in order* on exit from the scope. The semantics of sieve scopes can be considered as generalising to composite statements the semantics of the Fortran 90 single-statement vector assignments [17]. This semantics, named call-by-value-delay-result (CBVDR) [15], disallows flow dependences and preserves name dependences on data in global memory, and by restricting dependence analysis to data in local memory makes C++ code more amenable to automatic parallelisation.

8 Conclusion and Future Work

We have presented the concept of decoupled Access/Execute specifications and demontrated their convenience, flexibility and efficiency on three benchmark examples. Our Æcute implementation automates the data movement element of the accelerator programming task. The blocking of DMA transfers and construction of DMA lists enabled by separating the memory access from computation results in more efficient memory traffic.

We are looking into extending this work in several ways.

First, Æcute specifications may be thought of at a level of compiler intermediate representation rather than a high level programming language. Thus, we plan to investigate 'front-ends' that will derive Æcute specifications from higher-level abstractions, in particular, from the polyhedral model [18]. In addition, we wish to investigate 'back-ends' for other accelerator architectures, such as GPUs.

Second, we plan to integrate Æcute specifications into a compiler, to reduce both the overhead of interpreting Æcute specifications at run-time and the size of generated data-movement code, which must be minimised to conserve precious local memory. As in Gaster's streaming extension to OpenMP [19], compiler support can be layered on top of an extention and streamlining of the current Æcute classes, allowing the application to work correctly with or without compiler support.

Adding compiler support is related to our work on metadata-enhanced component programming [20], which uses Æcute-like metadata, describing the input-output interfaces of components, such that combining the components can optimise data flow at run-time. We aim to achieve similar optimisations by applying fusion optimisations to Æcute kernels.

Third, we plan to extend the expressivity of Æcute metadata to handle a larger set of kernels, associated with full scale applications. The current Æcute implementation supports only a limited range of partitioning options and mappings to data. We can extend this by using a hierarchical partitioning and improving the search options, *e.g.* for locality. In addition, we wish to support unstructured mesh based computations, such as fluid flow. For unstructured data we need to extend the memory read and write sets to support indirection while maintaining decoupling of access and execute.

Acknowledgements. We thank Mike Gist for his implementation of matrix-vector multiply, the anonymous reviewers and Alec-Angus Macdonald for their helpful comments, and the EPSRC for funding this work through grant number EP/E002412/1.

References

1. Hofstee, H.P.: Power efficient processor architecture and the Cell processor. In: Proceedings of the 11th International Conference on High-Performance Computer Architecture (HPCA), pp. 258–262. IEEE Computer Society, Los Alamitos (2005)
2. ClearSpeed Technology: The CSX architecture, `http://www.clearspeed.com/`
3. Smith, J.E.: Decoupled access/execute computer architectures. ACM Trans. Comput. Syst. 2(4), 289–308 (1984)

4. Watson, I., Rawsthorne, A.: Decoupled pre-fetching for distributed shared memory. In: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS), Washington, DC, USA, pp. 252–261. IEEE Computer Society, Los Alamitos (1995)
5. Tremblay, M., Chaudhry, S.: A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC processor. In: Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC) (2008)
6. Topham, N., Rawsthorne, A., McLean, C., Mewissen, M., Bird, P.: Compiling and optimizing for decoupled architectures. In: Proceedings of Supercomputing (SC), p. 40 (1995)
7. Lau, D.L., Gonzalez, J.G.: The closest-to-mean filter: an edge preserving smoother for Gaussian environments. In: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 2593–2596. IEEE Press, Los Alamitos (1997)
8. Warren, H.S.: Hacker's Delight. Addison-Wesley, Boston (2002)
9. Carter, L., Gatlin, K.S.: Towards an optimal bit-reversal permutation program. In: Proceedings of Foundations of Computer Science (FOCS), pp. 544–555 (1998)
10. Wright, C.: IBM software development kit for multicore acceleration. Roadrunner tutorial LA-UR-08-2819 (2008), `http://www.lanl.gov/orgs/hpc/roadrunner`
11. Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.: Sketching stencils. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI), pp. 167–178. ACM, New York (2007)
12. Saltz, J.H., Mirchandaney, R., Crowley, K.: Run-time parallelization and scheduling of loops. IEEE Trans. Comput. (5), 603–612 (1991)
13. Fatahalian, K., et al.: Sequoia: programming the memory hierarchy. In: Proceedings of Supercomputing (SC), pp. 83–92 (2006)
14. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: Proceedings of Supercomputing (SC), pp. 86–96 (2006)
15. Lokhmotov, A., Mycroft, A., Richards, A.: Delayed side-effects ease multi-core programming. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 641–650. Springer, Heidelberg (2007)
16. Codeplay Software: Portable high-performance compilers, `http://www.codeplay.com/`
17. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, San Francisco (2002)
18. Griebl, M.: Automatic Parallelization of Loop Programs for Distributed Memory Architectures. University of Passau, Habilitation Thesis (2004)
19. Gaster, B.R.: Streams: Emerging from a shared memory model. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 134–145. Springer, Heidelberg (2008)
20. Howes, L.W., Lokhmotov, A., Kelly, P.H., Field, A.J.: Optimising component composition using indexed dependence metadata. In: Proceedings of the 1st International Workshop on New Frontiers in High-performance and Hardware-aware Computing (2008)

MPSoC Design Using Application-Specific Architecturally Visible Communication

Theo Kluter, Philip Brisk, Edoardo Charbon, and Paolo Ienne

Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
{ties.kluter,philip.brisk,edoardo.charbon,paolo.ienne}@epfl.ch

Abstract. This paper advocates the placement of *Architecturally Visible Communication* (AVC) buffers between adjacent cores in MPSoCs to provide high-throughput communication for streaming applications. Producer/consumer relationships map poorly onto cache-based MPSoCs. Instead, we instantiate application specific AVC buffers on top of a distributed consistent and coherent cache-based system with shared main memory to provide the desired functionality. Using JPEG compression as a case study, we show that the use of AVC buffers in conjunction with parallel execution via heterogeneous software pipelining provides a speedup of as much as 4.2x compared to a baseline single processor system, with an increase in estimated memory energy consumption of only 1.6x. Additionally, we describe a method to integrate the AVC buffers into the L1 cache coherence protocol; this allows the runtime system to guarantee memory safety and coherence in situations where the parallelization of the application may be unsafe due to pointers that could not be resolved at compile time.

1 Introduction

The memory and communication architectures proposed for current and next generation multi- and many-core MPSoCs do not match the needs of streaming applications. Streaming applications use the *Synchronous Data Flow (SDF)* model of computation [11], in which coarse-grained communication is modeled as a pipeline; this pipeline, in turn, can be viewed as the concatenation of a set of producer/consumer relationships. The performance of streaming applications highly correlates with the ability of the pipeline to effectively (1) overlap computation with communication and (2) balance the workload across the multitude of cores in the system. The non-determinism imposed by packet-based on-chip networks is detrimental; likewise, buses and crossbars do not support concurrent communication and quickly saturate as the number of cores increases. Furthermore, cache-to-cache communication is a bottleneck, as the concurrent transfer of data from producer to consumer leads to an excess of coherence traffic within the memory system. These sources of overhead suggest that alternative MPSoC interconnect is required to support streaming applications.

The ideal communication architecture for a producer/consumer relationship is a double-buffer placed between the producer and the consumer. The buffer is replicated so that the producer can write to one buffer as the consumer concurrently reads from

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 183–197, 2009.

the other; this enforces the safety property that the producer cannot overwrite the consumer's data before it is read. When the producer and consumer are finished writing and reading respectively, they swap buffers and repeat the process. This communication architecture has already been used in Streamroller [10], a high-level synthesis system for streaming applications; it also bears some principle similarities to the FIFO interface of the Tensilica LX2 processor [18].

This paper advocates the instantiation of application-specific double buffers between adjacent cores in MPSoCs in order to enhance memory system performance for stream programs. We refer to each double buffer as a single *Architecturally Visible Communication (AVC)* buffer. AVC buffers can be viewed as a scratchpad memory that is shared between two cores.

AVC buffers offer several distinct advantages over on-chip networks and bus-based communication schemes: (1) communication is deterministic; network congestion and bus saturation are wholly eliminated; (2) computation and communication are effectively overlapped; compiler techniques to optimize load-balancing across cores already exist; (3) each access to the AVC buffer is cheaper than a cache access: since there is no tag array, and only one way (direct mapped), each access to the AVC buffer consumes less energy and takes fewer cycles than a cache access; and (4) cache coherence traffic for producer/consumer data is eliminated, reducing pressure on the memory subsystem.

At present, automatic parallelization methods [14,19] are not safe for streaming applications written in languages such as C that permit arbitrary pointer arithmetic. Profile-based algorithms are dependent on the dataset used, and therefore may not uncover all data-dependencies. It is conceivable that a pointer that could not be resolved by the compiler may attempt to modify the contents of the AVC buffer. Furthermore, the processor that executes this access may not be the producer or consumer of the data in question. This access cannot execute, as the data has been statically removed from the memory system by the compiler. To ensure program correctness, in the rare event that a undiscovered data-dependence does occur during runtime, a safety engine is invoked to dynamically resolve the coherence problem by removing data from the AVC buffer and reinsert it into the memory system. Although this degrades performance, it may be necessary to ensure the correctness of the program across all possible executions.

The remainder of the paper is organized as follows: Section 2 details the related work in the domain. Section 3 discusses the specific problems that one could encounter in the introduction of AVC buffers inside an MPSoC, and brings effective and efficient solutions to all of them. We prove this in Section 4.2 by addressing a complete application displaying all qualitative situations of interest, and by using the experimental environment described in Section 4.1. Section 5 concludes the paper.

2 Related Work

In the Streamroller high-level synthesis system [10], high-throughput pipelines are synthesized for streaming applications written in C; AVC buffers are placed between adjacent pipeline stages. Unlike in our work, computing elements are dedicated loop accelerators rather than processors. The Tensilica LX2 processor [18] allows the user to instantiate FIFOs between communicating processors, similar in principle to our AVC

buffers. Our AVC buffers, in contrast, are more general: the producer (consumer) may write (read) the data from its buffer in any order, i.e., FIFO semantics are not imposed on the communication medium.

One of the challenges is to compile sequential applications written in high-level languages, such as C/C++. Rul et al. [14] developed heavyweight, unsafe profile-based methods to identify communication patterns in sequential programs. This method is unsafe, as some communication patterns that are theoretically possible may not be induced to occur by the input data used to collect the profile. The parallelization operates under the assumption that inter-processor communication is expensive, so the authors attempt to map producers and consumers onto the same target architecture. In this work they introduce homogeneous, and heterogeneous software pipelining, where (1) in homogeneous software pipelining, each processor executes all kernels on a subset of the input dataset in data parallel fashion, and (2) in heterogeneous software pipelining, each processor executes one single kernel, and applies it to the complete input dataset. The study of bzip2 in [14] concludes that homogeneous software pipelining is superior to heterogeneous software pipelining; our results show that the inclusion of AVC buffers leads to the opposite conclusion.

Thies et al. [19] described a semi-automated method to identify coarse-grained pipeline parallelism in programs written in C. Similarly to our work, the authors concluded that heterogeneous pipelining is superior to homogeneous pipelining for streaming applications. Their method, however, requires that the programmer annotate the code with explicit pipeline boundaries. In principle, their method could be made aware of AVC-buffers and it could target a system such as ours. As our system is application-specific, we give the programmer/compiler the freedom, in principle, to chose AVC buffers of the appropriate size.

Streaming languages, such as StreamIt [2] are dedicated to streaming applications; due to their favorable semantics, numerous compiler optimizations have been proposed, many of which are relevant to this work. Sermulins et al. [15], in the context of a compiler for a single processor system, argued in favor of heterogeneous pipelining that has many similarities to ours. Consider a simple pipeline with two stages, S and T. One approach is to use an ordering ordering STSTST... for pipeline stage invocation. If the instruction cache (I-cache) is large enough to hold S or T, but not both, then each invocation would cause a miss in the I-cache. On the other hand, a ordering similar in principle to homogeneous pipelining would yield an invocation order of SS....STT....T, where S is invoked N times followed by N invocations of T. The only I-cache miss would occur when the ordering transitions from S to T. The drawback is that S would produce N times as much data before T could start consuming it; in principle, this could lead to data cache misses. Their compiler, heuristically attempts to find the best invocation ordering to minimize the aggregate overhead due to cache misses. What is important to note is that the homogeneous pipeline organization creates the I-cache problem, as each processor must execute each stage of the pipeline; heterogeneous pipelining, in contrast, does not suffer from this drawback, as just one pipeline stage executes on each processor through the duration of the application.

Several processor architectures for streaming applications have been proposed in recent years. The Imagine [1], Storm-1 [9], and Merrimac supercomputer [3] employ

wide SIMD pipelines that are fed by local register files (LRFs); a very wide streaming register file (SRF) is placed between the LRFs and memory, leading to an effective overlap of computation and communication. The SODA architecture for software-defined radio [13] contains a 32-way wide SIMD pipeline, a simpler scalar pipeline, and an address generation pipeline; communication between these pipelines is accomplished through scratchpad memories, with shuffle exchange networks to assist with serial-to-parallel and parallel-to-serial conversion, which occurs at critical points in their application domain. The Raw microprocessor [17] uses an a scalar operand on-chip network to route streaming data between adjacent functional units; delays, which vary based on the relative placement of tasks, are exposed to the compiler. Our MPSoC is most similar to Raw, as the processors themselves are simple 5-stage RISCs, without support for SIMD operations and wide register files; however, our use of AVC buffers is similar to the SRF/LRF and scratchpad memories, but for processor-to-processor communication.

Several compiler optimization methods for streaming programs targeting streaming architectures have been proposed in recent years. Das et al. [4] focused on techniques to effectively map data to the SRF, along with strip mining, loop unrolling and software pipelining. Lin et al. [12] described a hierarchical modulo scheduling algorithm targeting SODA, which used a general solver tool based on *Satisfiability Modulo Theory (SMT)*. These methods are tied to specific target processor architectures and attempt to exploit features that are not present in our system.

Gordon et al. [6] developed several compiler optimizations targeting the Raw microprocessor. Their work focuses on splitting and fusing tasks within the streaming application's intermediate representation, and then mapping them onto different processors in the system. Their abstract model of communication is a FIFO, similar in principle to the Tensilica LX processor feature described above [18]; however, it must be mapped onto the resources of the target machine: in this case, Raw's scalar operand network. In a follow-up paper, Gordon et al. [5] refined their splitting and fusing to account for both task and data parallelism, and introduced a new method for coarse-grained software pipelining; they achieved an average improvement of 1.87x over their previous work. In principle, these methods could easily be adapted to exploit the AVC buffers; the development of automatic parallelization and compilation techniques targeting MPSoCs with AVC buffers is left open for future work.

3 The Importance of Inter-processor Communication

This section begins with the assumption that all data dependencies can be resolved at compile-time, including pointers. We show how the proposed ideas can be generalized for the cases where all data-dependencies cannot be resolved.

3.1 Producer/Consumer Relationships

A producer/consumer relationship can be qualified as the sharing of a data structure between two or multiple kernels/functions in a program. The shared data structure is in its simplest form a scalar, but can also be a multidimensional array. In this work we assume the shared data structures to be only a fraction of the size of the data caches. The

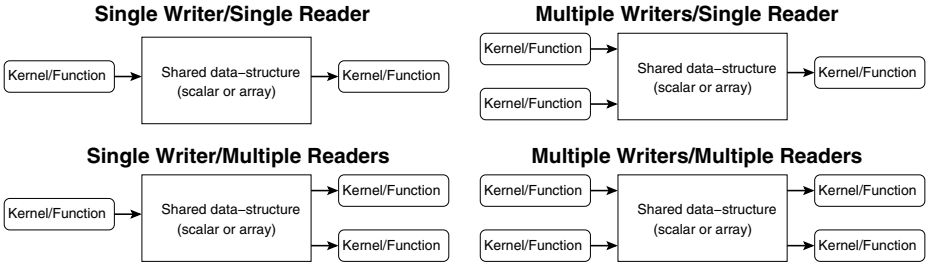


Fig. 1. The four different types of producer/consumer relationships

rationale behind this assumption is that (1) excessively large data structures would lead to extensive capacity misses inside the data-cache, thus increasing energy consumption and impeding performance and (2) they would require large buffers (memory), thus potentially increasing the processor's critical path (see Section 3.5), and would consume excessive silicon real estate. Producer/consumer relationships can be divided into four different types as shown in Figure 1. Note, however, that a *producer* may, during its execution, also read from the data structure; a *consumer* may also write to the data structure during its execution. Additionally producer/consumer relationships can be classified based on their access patterns (1) sequential access patterns occur when the elements of the data structure are accessed in increasing/decreasing address order; (2) random access patterns occur when the order of accessing the elements of the data structure is random or cannot be resolved at compile time.

3.2 JPEG Compression Algorithm

For the remainder of the paper we use the JPEG compression algorithm as motivational example and case study. Although JPEG compression is a relatively simple algorithm, it is easy to understand, and representative for streaming applications.

Figure 2 shows the block diagram of the JPEG compression algorithm. The top of Figure 2 shows the four kernels of the JPEG compression algorithm; the bottom shows a schematic data-flow representation of the algorithm. The JPEG compression algorithm contains four major producer/consumer relationships. The first three are between the four kernels, and require a buffer containing 8x8 16-bit values. The fourth one is not explicitly visible in Figure 2, as this relationship is between two consecutive entropy-encoding steps. As the JPEG compression algorithm uses *differential DC-component compression*, the entropy-encoding kernel has a producer/consumer relationship with itself in form of three 8-bit scalars.

The access patterns can be quantified as: (1) sequential writes by the color space conversion and quantization kernels; (2) random reads/writes by the discrete cosine transformation and entropy encoding kernels. All producer/consumer relationships are *single producer/single consumer*, except for the *Discrete Cosine Transformation (DCT)* kernel. As the DCT-kernel performs its operations first column-wise, and then row-wise, it can be viewed internally as a *multiple producer/multiple consumer* relationship.

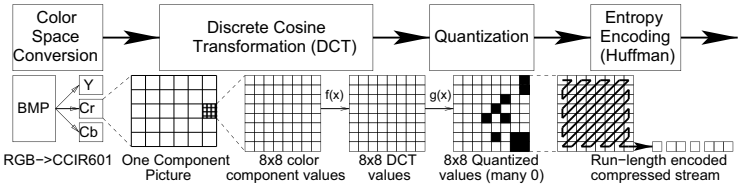


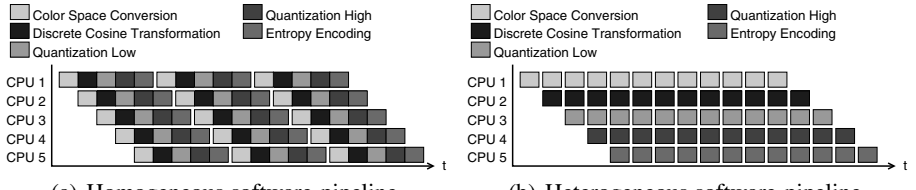
Fig. 2. The Block diagram of the JPEG compression algorithm as motivational example

3.3 Parallelization

In this work we apply synchronous software pipelining with static scheduling to perform parallelization. The synchronous nature of the pipelining is achieved by applying hardware-barrier synchronization between the kernels of the program. A *barrier* can be seen as a global *clock* to the system. Each processor can proceed from the barrier if and only if all other processors have entered it. Contrary to synchronous software pipelining, asynchronous software pipelining synchronizes by *completion detection* in the form of locks/semaphores, and is beyond the scope of this work.

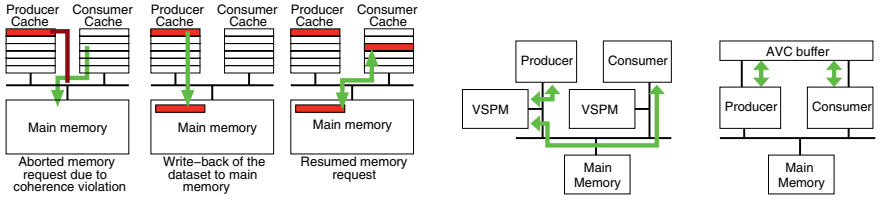
In software pipelining there are basically two styles: homogeneous, and heterogeneous pipelining. Figure 3 shows the two software pipelining styles applied to the JPEG compression algorithm. Homogeneous software pipelining keeps the producer/consumer relationships within the data cache of each processor; each processor, however, must store the complete program in its instruction cache. Heterogeneous software pipelining, distributes the program code across the instruction caches of the different processors without replication; moreover, it exposes the producer/consumer relationship to the memory subsystem. The trade-off of which software pipelining style to use therefore lies in the overhead induced by capacity misses in the cache versus producer/consumer relationships exposed to the memory subsystem. As the cost of exposing producer/consumer relationships is higher than the capacity-miss overhead (as discussed in Section 3.5), most parallelization algorithms tend to prefer homogeneous over heterogeneous software pipelining.

In our system, parallelization and the scheduling of kernels on processors is performed statically at compile time; each kernel will have a direct connection to the appropriate AVC buffer(s).



(a) Homogeneous software-pipeline. (b) Heterogeneous software-pipeline.

Fig. 3. The two software pipelining styles applied to the JPEG compression algorithms



(a) The different stages in a memory-subsystem exposed producer/consumer relationship in presence of a coherence protocol.

(b) The Virtual Scratchpad Memory and AVC buffer solution to memory-subsystem exposed producer/consumer relationships.

Fig. 4. The exposure of producer/consumer relationships in three different memory subsystems

3.4 Coherence and Consistence

The introduction of distributed memory elements in a shared-memory system leads to the situation where multiple-copies of the same data are dispersed throughout the system. If these copies are read-only there is no problem; however if one of these copies is assigned a new value then potential coherence and consistence violations may occur.

The sequential consistence model states that all read and write operations are observed as atomic, and in program order. Our programming paradigm is based on this consistence model. The consistence model is primarily enforced in software, but may also be hardware-assisted. In this work we assume software-enforced consistence.

The coherence rule is less strict than the consistence rule, as it only requires all write operations to be seen as atomic operations. In other words, a read operation should always see the latest written value to the shared-memory, regardless of where the write occurs. Coherence, in general, is enforced by the compiler in scratchpad memory-based systems and by hardware in cache-based systems.

3.5 AVC-Buffers, Caches, and ScratchPad Memories

In a single processor system, producer/consumer relationships are generally hidden from the memory subsystem. Although the shared data structures are allocated in main memory, they will almost never leave the data-cache due to (1) explicit locking of the cache-lines in which the shared data structures reside [7], or (2) implicit locking of the cache-lines in which the shared data structures reside by exploiting the temporal locality of the data structure by using, for example, a *Least Recently Used* replacement policy in the cache. In the rare case, when no inter-processor communication exists (e.g., completely data-parallel algorithms), the same technique can be applied to cache-based MPSoCs. Streaming applications, however, require inter-processor communication, and a coherence problem arises (see Section 3.4). An example of inter-processor communication is an exposed producer/consumer relationship between two processors. To insulate the programmer/compiler from the coherence problem, most MPSoCs are provided with a hardware coherence protocol. Most prominent hardware coherence protocols are snoopy protocols with three or more states. The simplest snoopy protocol is the *Modified, Shared, and Invalid (MSI)* states based protocol. More sophisticated protocols utilize *MESI* or *MOESI* states. It is beyond the scope of this work to describe the

details of these protocols; however it is important to note that they severely impact on the overhead induced due to exposed producer/consumer relationships. In this work we utilize the most prominent *MESI* states-based hardware coherence protocol.

To understand the cost involved when a producer/consumer relationship is exposed to the memory subsystem, we consider a *single producer/single consumer* relationship as shown in Figure 1. The different stages involved in the coherence protocol are shown in Figure 4(a). The communication starts by the consumer cache making a request (read or write) to the memory subsystem for a shared data structure—as shown to the left in Figure 4(a). The consumer's cache will request the data structure, which is assumed to exist in a modified state in the producer's cache; thus, the data structure is invalid in main memory. Next the producer's cache will abort the consumer's request, as it holds the latest copy. As a reaction to this request/abort action, the producer's cache will write back the shared data structure to main memory as shown in the middle of Figure 4(a). Finally the consumer's cache will resume the memory request and copies the "correct" data structure to itself from main memory, and execution can continue—shown to the right of Figure 4(a). During the write-back stage, the producer is likely to stall (impeding the producer's performance), whilst the consumer is likely to stall during the whole transfer. Furthermore, extra energy is consumed, and bus bandwidth is expended by the write-back/read action to and from main memory. An improved version of on this scheme is to merge the write-back stage, shown in the middle of Figure 4(a), with the resumed memory request phase —shown to the right of Figure 4(a). This optimization gives the consumer the possibility to "read" the values of the data structure during the write-back phase. This optimization will be referred to as a *cache-to-cache copy*.

The scratchpad memory approach is different than that of a cache-base system, as it provides coherence at compile-time. In *Virtual Shared ScratchPad Memory (VSPM)* systems, each processor can access each of the scratchpads at different costs. To deal with shared data structures in conjunction with the coherence problem, the compiler places the shared data structure solely in one scratchpad memory. Coherence is guaranteed, and less energy is consumed, as the data structure is not copied to main memory. Arguably, the impact on performance, in comparison to a cache-based system is equal, or even higher, as each access to a remote scratchpad memory is transmitted on the bus; direct access to a cache is much faster—shown in the left of Figure 4(b). Also the impact of bus bandwidth is at least as large as the cache-based system, as remote scratchpad accesses require bus transactions.

AVC buffers, as shown on the right of Figure 4(b), completely remove the producer/consumer traffic from the memory subsystem. AVC buffers benefit both from this removal, and from the fact that the buffers are moved forward in the processor pipeline using the *Instruction Set Extension (ISE)* interface of the processor. This has a significant impact on the organization of the pipeline, as AVC buffer accesses occur during the execute stage of the pipeline, rather than the write-back stage. This ensures that the AVC buffer load/store operations take a single cycle and are atomic. If the cache access takes multiple cycles (3 cycles for a hit, in our system), then the AVC buffer must spend an extra 3 cycles before it commits. If we have a store to the cache followed by an AVC buffer store, the AVC buffer store would commit and retire itself before the cache store commits and retires, violating consistence. Thus, number of cycles required to access

the AVC would need to be the same as the number required to access the cache. This consistence issue is wholly avoided by placing the AVC buffer at the execute stage of the pipeline; although the AVC buffer store occurs before the cache store finishes, the instructions are retired in-order, guaranteeing consistence.

Moving the AVC buffer load/store to the execute stage of the pipeline makes these operations single-cycle and atomic. The disadvantage is that the delay of the AVC-buffer might impact the processor's critical path if the buffer size is large (i.e., the memory read access time exceeds the processor's critical path delay). The AVC buffer does not impact the bus bandwidth, which increases performance, and reduces energy consumption per access compared to a cache and less to equal energy consumption per access compared to scratchpad memories.

3.6 Execution Safety

The discussion in Sections 3.1–3.5 assumed that all the data-dependencies of the shared data structures could be resolved at compile time or by the programmer. In this scenario, we can completely remove the data structure from the memory subsystem and move it into AVC-buffers; however, if we cannot resolve these data dependencies, we must ensure that the correct execution of the program is not jeopardized (we henceforth refer to the correct execution as *safe*).

In a scratchpad-based system all data structures with unresolved data-dependencies cannot be safely allocated to the scratchpad memory. When applying AVC-buffers in a scratchpad based system, we must take a similar approach, as otherwise coherence cannot be guaranteed and safety is jeopardized.

In cache-based systems, unlike scratchpad-based systems, coherence is dynamically enforced by the hardware coherence protocol. As our system contains caches as well as AVC buffers, we can use the hardware coherence protocol to allow data structures with unresolved data-dependencies to be candidates for AVC-buffer allocation. We will refer to these data structures with unresolved data-dependencies as *unsafe structures*. To guarantee the safety of unsafe structures, a three stage approach is taken. First, we allocate unsafe structures to both the AVC-buffers and main memory. Secondly, we transform the AVC-buffer into a coherence protocol-enhanced mini-cache. Finally, to prevent performance losses due to *false sharing*, we cache-block align all data structures.

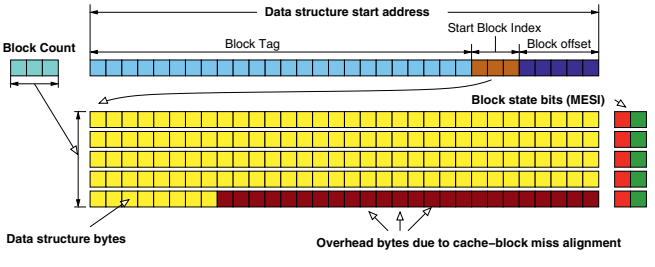


Fig. 5. The AVC buffer converted to a micro-cache

The impact of the redundant allocation of data in both AVC-buffers and memory, is minimal; we lose the advantage of freeing up main memory space by removing the unsafe-structures from it; however, in all other systems this double allocation occurs implicitly: the structure resides in both cache/scratchpad and memory.

The impact of the second action is more severe. The rationale behind the transformation is that in most cases the unsafe structure resides in the AVC-buffer in either the *exclusive* or *modified* state. In the rare case that it is requested by the memory subsystem it has to be reinserted into the memory sub-system by use of the coherence mechanism similar to coherent caches. As the reinsertion into the memory sub-system is rare, we can use a micro-cache architecture similar to a cache with segmented blocks. Our micro-cache, however, only contains one segmented cache line, one tag, and one pair of state-bits per segment (see Figure 5). Each segment of our cache line is the size of one coherence-unit of the applied data caches (i.e., the size of a block inside the data cache). The overhead of this approach is threefold: First, we introduce extra storage in form of a tag, block count, and block-state bits. Second, we must implement a complete cache controller with coherence protocol. Third, we can have overhead due to the fact that the unsafe structures size in the AVC buffer is not a multiple of the size of the coherence unit.

When the compiler allocates memory for all data structures, it may decide to place safe and unsafe data structures continuously in such a manner that the boundary between the data structures is not aligned with the boundaries of the coherence units. This possibly creates situations where the *overhead bytes*, as depicted in Figure 5, are occupied by other safe or unsafe data structures, in which there is a high chance of *false sharing*. *False sharing* is a well-known effect in which the safe data structure invalidates the unsafe structure, or vice versa; as both structures are independent this coherence traffic is redundant, reduces the AVC-buffer performance, and therefore should be avoided. *False sharing* can be avoided by coherence unit aligning all unsafe structures, which can be done automatically. The overhead bytes (as shown in Figure 5) do not require memory elements in the AVC buffers due to cache-block alignment.

It should be noted that this coherent issue does not occur when programs are written using streaming languages, such as StreamIt [2]. These languages do not support arbitrary pointer arithmetic, and as a result, safety violations of this kind cannot occur. Thus, this safety mechanism is only required if the application is written in an inherently unsafe language, such as C.

4 Experiments

4.1 Experimental Setup

We implemented our AVC buffers by augmenting an OpenRISC-compatible platform running on FPGA. The AVC buffers are coupled to the processors utilizing their *Instruction Set Extension (ISE)* interface, where the extended instructions are solely AVC-buffer load/store instructions. Furthermore, for the multi-processor case, we augmented the architecture with a hardware barrier.

We parallelized the *cjpeg* program from the *EEMBC* denbench suite [8]. The code has been parallelized by hand, while keeping in mind that automatic parallelization

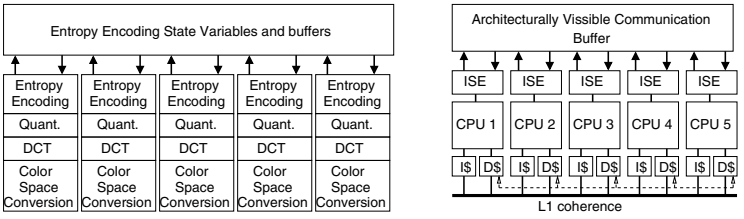
algorithms as presented in Section 2 may be able to perform the job as well. Care has been taken to avoid *false sharing* by aligning all data structures on cache line boundaries. The parallelized versions of the JPEG compression are statically mapped onto the 5 processor system. Finally, the complete code-base has been cross-compiled using a "newlib"-based gcc 3.4.4 tool-chain for the OpenRISC.

For all the experiments we used the same 24-bit RGB encoded picture of 1024x768 pixels, similar to the resolution of current high-end web-cams and standard portable phones. For the energy consumption calculations we used CACTI [16] to determine the read/write energy-consumption for different cache configurations in a 90 nm technology The external memory and bus-access read/write energy consumption is estimated to be $792pJ$ per access. The energy values reported here only include the dynamic energy consumed in the memory sub-system; this model does do not include processor and leakage energy.

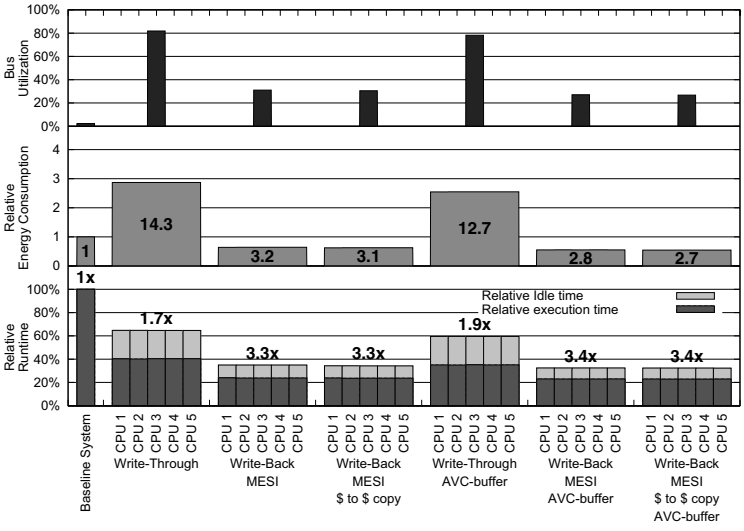
4.2 Experimental Results

To enable a fair comparison, we performed a performance-energy exploration of a single processor-based system running the JPEG compression algorithm. Our baseline architecture was the one that performed best with the minimal energy consumption; it used a 4 kB direct mapped instruction cache, and an 8 kB 2-way set-associative data cache. Next we analyzed the runtime of the different kernels of the JPEG compression algorithm on the baseline architecture. This runtime breakdown guided the creation of a heterogeneous software pipelined five processor architecture shown in Figure 6(a), and a homogeneous version shown in Figure 7(a). Each of the processors of the two systems uses a 4 kB direct mapped instruction cache, and an 8 kB 2-way set-associative data cache with Level 1 MESI-states hardware coherence protocol.

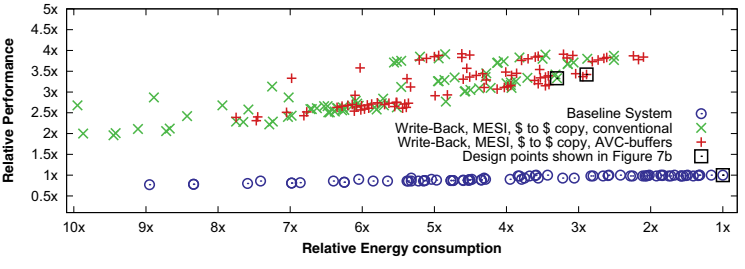
We first accelerate the application by employing a write-through coherence policy, which does not require a hardware coherence protocol implementation. As to be expected, on both the homogeneous and heterogeneous versions the bus is completely saturated—as shown in Figure 6(b) and Figure 7(b), limiting the speedup to a factor of 1.7x compared to the baseline system. Next we accelerate the application on a five-core system that employs a write-back policy with the MESI protocol for cache coherence and snoopy cache-to-cache copies. When homogeneous pipelining is used, this system offers a speedup of 3.3x; the speedup achieved by the same system with AVC buffers is 3.4x compared to the baseline, a meager performance increase. Furthermore, as the inter-processor communication is only three 8-bit scalars (one cache-block), the influence of the cache-to-cache copy enhancement is minimal. When heterogeneous pipelining is used, on the other hand, the speedup of the five core system is 3.2x compared to the baseline, and increases to 4.2x through the addition of AVC buffers. Also the influence of the inter-processor communication is clearly visible. As in the heterogeneous case, the size of the communicated data is three arrays of sixty-four 16-bit values (four cache-blocks); the speedup of the system employing cache-to-cache copies is 3.2x, compared to a 3.0x speedup for the system without cache-to-cache copy; both systems enhanced with AVC-buffers show an equal speedup of 4.2x due to the removal of the inter-processor communication.



(a) Architecture, kernel mapping, AVC-buffer allocation, and MPSoC architecture.



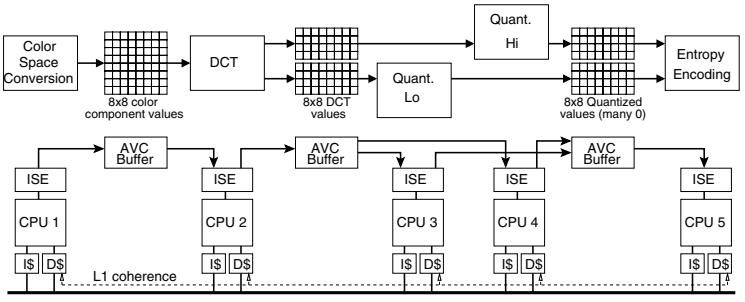
(b) Bus utilization, energy consumption and runtime of the different architectures compared to the baseline system.



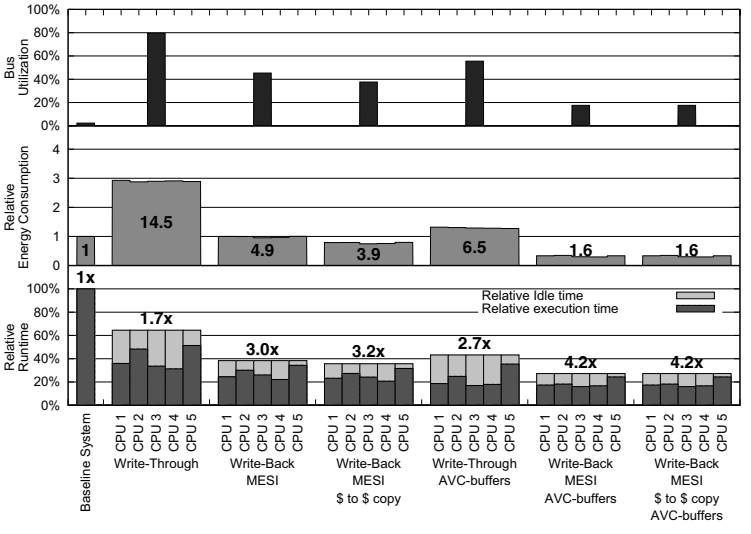
(c) Energy-performance exploration utilizing 2kB, 4kB, 8kB, direct-mapped, 2 way, and 4 way set-associative instruction and data caches.

Fig. 6. Homogeneous Software pipelining of the JPEG compression algorithm on a 5 processor MPSoC

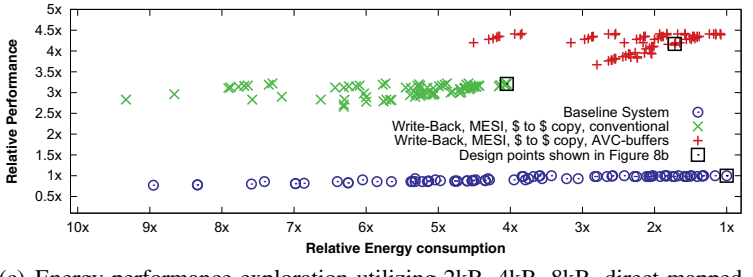
The use of AVC buffers in conjunction with heterogeneous pipelining is also beneficial in terms of its ability to reduce the energy consumption of the memory subsystem. Homogeneous pipelining without AVC buffers increased the memory subsystem energy consumption by 3.1x compared to the baseline single processor system; the inclusion



(a) Architecture, kernel mapping, AVC-buffer allocation, and MPSoC architecture.



(b) Bus utilization, energy consumption and runtime of the different architectures compared to the baseline system.



(c) Energy-performance exploration utilizing 2kB, 4kB, 8kB, direct-mapped, 2 way, and 4 way set-associative instruction and data caches.

Fig. 7. Heterogeneous Software pipelining of the JPEG compression algorithm on a 5 processor MPSoC

of AVC buffers reduced it to 2.7x. Heterogeneous pipelining without AVC buffers increased the memory subsystem energy consumption by 3.9x; however the inclusion of AVC buffers reduced it to 1.6x, which is quite low for a five core system.

Finally to ensure that the presented results are not biased by a poor choice of caches, we performed an exhaustive energy-performance exploration for the cache-to-cache copy enhanced write-back MPSoCs with and without AVC-buffer extension. The results of this exploration is shown in Figure 6(c) and Figure 7(c). Both figures show that: (1) the results are consistent for all cache configurations; (2) the homogeneous software pipelining clearly suffers cache pressure as described in Section 3.3; (3) the heterogeneous software pipelining, originally suffering memory-subsystem pressure due to inter-processor communication, performs consistently better by adding AVC-buffers. Thus we conclude that heterogeneous pipelining with AVC buffers is the best communication architecture for our five core MPSoC implementation of the JPEG compression.

5 Conclusion

This paper discusses a case study using JPEG compression that motivates the use of Architecturally Visible Communication buffers to accelerate producer/consumer communication in MPSoCs for streaming applications. Previous work on automated parallelization has favored homogeneous over heterogeneous software pipelining due to the high cost of core-to-core communication via the memory system. Because of this high communication cost, the most efficient pipelining method mapped producers and consumers of the same data onto the same core; however this approach does not effectively overlap computation and communication, which is of great importance when accelerating streaming applications. Our results show that the inclusion of Architecturally Visible Communication buffers yields the opposite result: providing a fast core-to-core communication mechanism favors heterogeneous over homogeneous software pipelining; these results were consistent and robust over a wide variety of cache configurations.

Automated techniques to parallelize applications written in C are unsafe, as pointer resolution is undecidable in the general case. For our work, the implication of the lack of safety is that data structures that have been removed from the memory subsystem and placed into Architecturally Visible Communication buffers can, theoretically, be accessed by a pointer. Our safety mechanism transforms the Architecturally Visible Communication buffer into a small cache that is connected to the coherence protocol; when an extraneous pointer accesses data in the Architecturally Visible Communication buffer, it is moved back into the memory system; although this implies some performance overhead, the occurrence of such pointer accesses is rare, thus mitigating its impact on the performance of the system, while ensuring correctness and safety.

References

1. Ahn, J.H., et al.: Evaluating the imagine stream architecture. In: Proceedings of the 31st Annual International Symposium on Computer Architecture, Munich, Germany, pp. 14–25 (2004)
2. Amarasinghe, S., et al.: Language and compiler design for streaming applications. International Journal of Parallel Programming 33, 261–278 (2005)

3. Dally, W.J., et al.: Merrimac: Supercomputing with streams. In: Proceedings of the Fifteenth International Conference on Supercomputing, Phoenix, Arizona, pp. 35–42 (November 2003)
4. Das, A., Dally, W.J., Mattson, P.: Compiling for stream processing. In: Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques, Seattle, Washington, pp. 33–42 (September 2006)
5. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, pp. 151–162 (October 2006)
6. Gordon, M.I., et al.: A stream compiler for communication-exposed architectures. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, pp. 291–303 (October 2002)
7. Gummaraju, J., Rosenblum, M.: Stream programming on general-purpose processors. In: Proceedings of the 38th Annual International Symposium on Microarchitecture, Barcelona, Spain, pp. 343–354 (November 2005)
8. Halfhill, T.R.: EEMBC releases first benchmarks. Microprocessor Report (May 1, 2000)
9. Khailany, B.K., et al.: A programmable 512 gops stream processor for signal, image, and video processing, vol. 43, pp. 202–213. IEEE, Los Alamitos (2008)
10. Kudlur, M., Fan, K., Mahlke, S.: Streamroller: Automatic synthesis of prescribed throughput accelerator pipelines. In: Proceedings of the 14th International Conference CODES-ISSS, Seoul, Korea, pp. 270–275 (October 2006)
11. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. 36(1), 24–35 (1987)
12. Lin, Y., et al.: Hierarchical coarse-grained stream compilation for software defined radio. In: Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, Salzberg, Austria, pp. 115–124 (September 2007)
13. Lin, Y., et al.: Soda: A low-power architecture for software-defined radio. In: Proceedings of the 33nd Annual International Symposium on Computer Architecture, Boston, Massachusetts, pp. 89–101 (June 2006)
14. Rul, S., Vandierendonck, H., de Bosschere, K.: Detecting the existence of coarse-grain parallelism in general-purpose programs. In: Proceedings of the 1st Workshop on Programmability Issues for Multi-Core Computers, Goteborg, Sweden (January 2008)
15. Sermulins, J., et al.: Cache aware optimization of stream programs. In: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, Chicago, Illinois, pp. 115–126 (June 2005)
16. Tarjan, D., Thoziyoor, S., Jouppi, N.P.: CACTI 4.0. Technical Report HPL-2006-86, Hewlett-Packard Development Company, Palo Alto, Calif. (June 2006)
17. Taylor, M.B., et al.: Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams. In: Proceedings of the 31st Annual International Symposium on Computer Architecture, Munich, Germany, pp. 2–13 (June 2004)
18. Tensilica. Xtensa LX2: Product Brief (April 2007)
19. Thies, W., Chandrasekhar, V., Amarasinghe, S.: A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In: Proceedings of the 40th Annual International Symposium on Microarchitecture, Chicago, Illinois, pp. 356–359 (December 2007)

Communication Based Proactive Link Power Management*

Sai Prashanth Muralidhara and Mahmut Kandemir

Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA 16802, USA
{smuralid,kandemir}@cse.psu.edu

Abstract. As the number of cores in CMPs increases, NoC is projected to be the dominant communication fabric. Increase in the number of cores brings an important issue to the forefront, the issue of chip power consumption, which is projected to increase rapidly with the increase in number of cores. Since NoC infrastructure contributes significantly to the total chip power consumption, reducing NoC power is crucial. While circuit level techniques are important in reducing NoC power, architectural and software level approaches can be very effective in optimizing power consumption. Any such technique power saving technique should be scalable and have minimal adverse impact on performance. We propose a dynamic, communication link usage based, proactive link power management scheme. This scheme, using a Markov model, proactively manages communication link turn-ons and turn-offs, which results in negligible performance degradation and significant power savings. We show that our prediction scheme is about 98% accurate for the SPEC OMP benchmarks and about 93% over all applications experimented. This accuracy helps us achieve link power savings of up to 44% and an average link power savings of 23.5%. More importantly, it incurs performance penalties as low as 0.3% on average.

1 Introduction

Power inefficiency coupled with limited instruction level parallelism changed the trend from increasing single core frequencies to having multiple relatively simpler cores on a single chip. Driven by this need to have power efficient systems, these chip multiprocessors (CMPs) have become the order of the day [26] [12] [35] [3]. With the projected increase in the number of cores in CMPs [35], limited scalability of bus structures and the need for more on-chip communication bandwidth have become major issues. These issues have given rise to network-on-chip (NoC) [6] [9] [11], which is a more scalable on-chip communication fabric.

The NoC framework addresses the scalability issue effectively. However, in such an NoC based CMP, the issue of power consumption can become a serious limiting factor. This is especially true since the power consumption is projected

* This research is supported in part by NSF grants 0811687, 0720645, 0720749, 0702519, 0444345 and a grant from GSRC.

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 198–215, 2009.

to increase rapidly as the size of NoCs increase. Therefore, there is a need to develop a wide variety of techniques to reduce chip power consumption.

A major contributor to chip power consumption is the NoC infrastructure. We found that, the NoC framework is responsible for as much as nearly 30% of the total chip power consumption. Communication links form a significant part of an NoC framework and their count increases with the number of cores in a CMP. This calls for power-aware design and power saving schemes which target not only power efficient cores but also power efficient link usage. Since, with the increase in the number of cores and with a similar increase in the number of communication links, possibility of more links being inactive increases dramatically, there is a need for a scalable power saving scheme which can exploit this effectively. Although circuit level and localized techniques are effective to an extent, they are not proactive, and therefore, lose out on important power saving opportunities. In this paper, we propose a completely proactive scheme aimed at link power management.

There have been significant research efforts aimed at characterizing the execution intervals of single-threaded applications into phases [29]. A program phase analysis is a technique of characterizing the program execution intervals into phases based on the similarity in their behavior. In the past, phase characterization has been used in the context of performance [29] [7] and power [15] [13]. We propose that execution of a multi-threaded application on an NoC based CMP can be characterized into phases based on the similarity across inter-core communication patterns. In this context, by communication pattern, we mean the usage of communication links in the system during execution. In case of a shared NUCA cache [17], which we consider, this usage of communication links is due to shared cache accesses and corresponding coherence traffic. The present circuit-level and localized schemes do not use this high level phase characterization information in their link power management. We propose to use the aforementioned phase characterization to implement a Markov based prediction scheme, which predicts the link usage of the next interval. This prediction can be used by a proactive link power management scheme to turn off predicted unused links and also to turn on links that are predicted to be used. The key advantage of this scheme is that, the links that are predicted to be used can be turned on ahead of time such that the turn-on latency is hidden and the performance remains unaltered. We show that this prediction based power management scheme can be very beneficial in reducing link energy consumption. We also note that this power saving scheme is remarkably scalable and can achieve increased power savings with increase in the number of on-chip cores and communication links.

We wish to note three other important points here. Firstly, one of the important goals of our scheme, apart from minimizing energy consumption, is also to minimize the adverse impact on performance. We later show that, our scheme is very accurate in predicting link usage and hence has almost negligible performance impact. Secondly, our scheme can work along with other circuit-level and localized hardware schemes to further maximize the benefits achieved. Thirdly, as we illustrate later, our scheme is highly scalable and is generic enough to be

adaptable across different NoC structures. To summarize, the main contributions of this paper are as follows:

• We classify the execution intervals of a multithreaded application executing on two-dimensional mesh based CMP into phases during runtime based on their similarity in communication link usage.
• We use this classification to implement a Markov prediction based, proactive link power saving scheme. More precisely, we show that a small prediction table can be maintained to make accurate predictions about the future communication link usage during runtime.
• We show that our prediction scheme is highly accurate achieving a prediction accuracy of over 98% in most applications and an average prediction accuracy of about 93% over all the applications we tested. As a result of this high prediction accuracy, the performance penalty incurred by our power management scheme is practically negligible, with an average value of 0.3%. Finally, we present the reduction in energy consumption, which is about 40% for two of the applications. We also present the average energy savings we achieve, which is about 23.5%.

The rest of the paper is organized as follows. Section 2 briefly summarizes the related work pertaining to our area. Section 3 provides a brief description of the NoC based CMP architecture we consider throughout this paper. Section 4 makes a case for a prediction based approach to link power optimization, and Section 5 provides a detailed description of phase classification based on link usage. Section 6 talks about the prediction based schemes we employ, and Section 7 provides a detailed description of how our power management scheme is implemented. Section 8 talks about the experimental setup, methodology, and results. Finally, we summarize and conclude with Section 9.

2 Related Work

With the growth of CMPs, there have been numerous efforts to optimize power in these systems both in the bus based architectures and the NoC based architectures. Isci et al analyze global dynamic power management policies for CMPs and propose dynamic schemes that perform better than static schemes [14]. Sharkey et al show that global power management outperforms local core-level schemes [28]. Li and Martinez present a scheme to dynamically optimize power consumption of a parallel application executing on a CMP under a given performance constraint [22]. There have been similar efforts to develop power management strategies in the context of NoCs. For example, Benini et al describe an energy-efficient interconnect framework design [4]. By comparison, Simunic et al use closed-loop control concepts to formulate a network-centric power management scheme [31]. There have also been prior research efforts targeting link power savings. Soteriou and Peh propose a dynamic power management scheme to turn off and turn on network links in a distributed fashion depending on network utilization [33]. Shin et al present a static scheme using voltage scalable links to optimize energy consumption [30]. Shang et al apply dynamic voltage

scaling to optimize link energy consumption [27]. In comparison, Li et al use profile information to implement a compiler based scheme which increases the link idle periods, thereby enabling the hardware schemes to be more effective in saving power [21]. They also propose a compiler-directed proactive scheme which analyzes the program during compile time and inserts link-activate and turn-off calls [20]. Our scheme differs from these link power management schemes in being a dynamic, runtime scheme which proactively turns off and turns on the interconnection links based on the prediction made by our online prediction module. Also, our dynamic scheme is more general and more widely applicable than compiler-based schemes which can be used only when the source code is available and statically analyzable.

Another related area of work is program phase analysis. Sherwood et al propose the concept of identifying repetitive execution intervals called phases using basic block vector (BBV) similarity [29]. Isci and Martonosi propose a phase analysis scheme for power and demonstrate that performance counter based schemes perform much better than control-flow based schemes such as those based on basic block vector (BBV) analysis when it comes to power [15]. There have been other attempts to use control flow information for program phase classification and consequent performance and power optimizations [13] [7]. Dhodapkar and Smith compare some of these techniques to detect program phases [8]. Perelman et al present a method to utilize phase analysis for parallel applications running on shared memory processors [25].

In this work, we use a flavor of phase characterization in our power management scheme but with important differences. First of all, we use inter-core communication as the basis for our phase characterization of multi-threaded applications. More precisely, intervals of execution which have similar communication patterns are characterized into a single phase. Secondly, we use a more fine grained form of phase analysis in our scheme, wherein, the length of instruction interval used is much shorter than those used in the prior schemes. This is because, we observed that, the inter-core communication, which essentially includes shared cache accesses and coherence traffic, exhibits repetitive behavior but only when looked at in shorter intervals. We further elaborate on this aspect in later sections. In essence, our goal is to identify repetitive behavior of a parallel application execution based on its inter-core communication pattern on the underlying NoC that connects the CMP nodes.

3 Target Architecture

We consider a two-dimensional mesh based NoC that connects the nodes of a CMP, although our approach is equally applicable to other NoC structures. In this architecture, each node (core) has a private level 1 (L1) cache. On the other hand, the level 2 (L2) cache is shared among all the cores and is banked with each core containing an L2 bank. Figure 1 shows a 4×4 mesh structure we use to convey our idea. Most of the time, unless otherwise mentioned, we consider this 16 core, 4×4 mesh based CMP with a shared L2 cache which is 16 banked

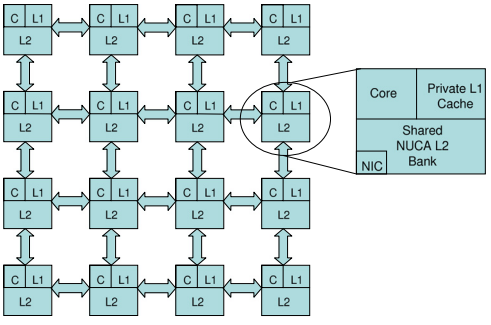


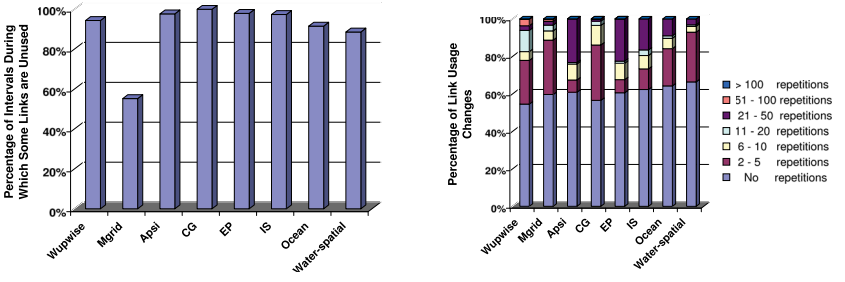
Fig. 1. A 4×4 mesh NoC based CMP. Note that this is a block diagram and not the actual layout, and the routers are not shown for clarity.

with each of the 16 cores containing an L2 bank. We use a static NUCA [17] scheme in this work although our scheme can be similarly used with dynamic NUCA [17] as well. We would like to emphasize that, in this paper, by "inter-core communication", we always mean an access made by a core to some other core's L2 bank.

4 Empirical Motivation

For any scheme aimed at link power savings to succeed, there should be considerable periods of execution during which some links are unused. If a multi-threaded application executing on an NoC based CMP uses all of the communication links during the entire period of execution, then any scheme aimed at saving link power will have limited returns. Fortunately, that is not the case in real applications. We profiled several parallel benchmarks from the SPEC OMP [2], NAS [1] and Splash2 [32] benchmark suites running on a 4×4 mesh architecture described in Figure 1. Profiling is done such that the execution is broken down into intervals of 5000 instructions, and links used during these intervals are recorded at the end of each such interval. We computed the percentage of such intervals during which at least some of the links in the interconnect network are not in use. Figure 2(a) shows our profiling results. As can be observed clearly, during a large percentage of intervals, at least some links are unused. Specifically, on average, in only 10% of intervals, all communication links are used. We also observed that the percentage increases slightly if the instruction interval is shortened. The number of links that are unused in such intervals determine the "window of opportunity", which in other words, means the amount of power savings that can potentially be extracted. The profiling results above serve as the key motivating factor for the scheme we propose in the coming sections.

Another key factor which needs to be considered is the "repetitive phase behavior" and hence possible "predictability" in parallel application's link usage. During execution, every time a new link usage pattern occurs, an important question is how long does that link usage pattern last before it changes again.



(a) (b)

Fig. 2. (a) shows the percentage of intervals during which at least a few links are unused. We see that, on average, in only about 10% of intervals, all links are used. (b) shows the number of intervals, a new link usage pattern lasts (repeats) before it changes again to a different usage pattern.

Figure 2(b) shows the distribution of the number of times a link usage pattern repeats before there is a change. On average, after 10% of link usage changes, link usage remains the same for 21 to 50 intervals. After 3% of link usage changes, the usage pattern remains the same for 11 to 20 intervals; after 6.6% changes, the same usage remains for 6 to 10 intervals and after 19.1% changes, 2 to 5 times. Overall, on average, whenever a new link usage pattern arises, on nearly 40% of occasions, it remains for more than one interval before it changes again. It is important to note that, we are talking about instruction intervals (intervals of 5000 instructions) here and hence the link usage pattern repeating twice implies that the link usage remains the same for 2×5000, which is for 10,000 instructions. This is an important statistic which hints at repetitiveness and predictability in link usage patterns and possible success of predictive schemes.

5 Link Usage Based Phase Classification

Repetitive behavior is an execution characteristic of most applications. This repetitive behavior can be on the basis of similarity in the basic blocks touched or on the basis of similarity in performance metrics such as cache misses [15]. We use inter-core communication as the basis for characterizing the program execution into phases. Therefore, we classify the execution intervals into phases based on communication link usage. Each execution interval is an interval of 5000 instructions in our classification scheme. Since communication pattern is an application characteristic, instruction interval can be customized for an individual application by using profiling results. Although this interval length can be configured and further tuned as mentioned above, we found that, an interval of 5000 instructions works well for all applications we tested since it captures the repetitive behavior in inter-core communication pattern well. The usage pattern of communication links during execution depends on the data allocations and the data access patterns exhibited by the application, which manifests itself as

L2 bank accesses. This means that, as the execution of a parallel application progresses, the L2 cache accesses and hence the communication link usage goes through phases. In this work, we represent the communication link usage in the form of a vector called "Link Vector", and carry out our phase characterization using this novel concept.

5.1 Link Vector

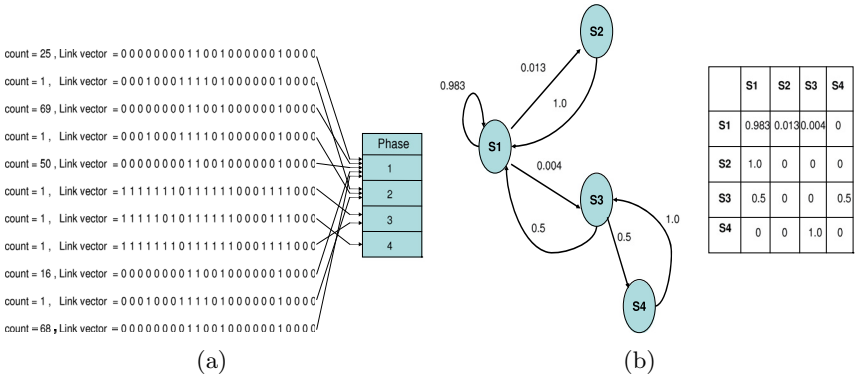
We represent the state of all the links in our NoC in the form of a link vector. Each bit in a link vector represents a link in the NoC and there is bit for every link. Consequently, the number of bits in the link vector is the same as the number in links in the on-chip network. Bit value 1 implies a used state, which means the link is being exercised, and a bit value of 0 implies an unused state, which means the link is idle. For example, in the case of NoC illustrated in Figure 1, the corresponding link vector contains 24 bits with each bit representing the current state of a link in the 4×4 mesh. The link vector of an execution interval is computed by ORing the link usage of all the instructions executed during the instruction interval. This essentially means that, even if a link is used only once during the entire interval, the link vector of the interval denotes that link as being used during the interval. Hence the motivation to have shorter instruction intervals when compared to considerably longer instruction intervals used in other phase characterization works [15] [29]. The effect of aforementioned scenario, where a link which is used only once during the interval and still being considered as used during the entire interval, has been minimized considerably by having shorter instruction intervals.

5.2 Runtime Classification

A simple way to identify phases is by using an identifier called "phase id" and a simple way to store phase information is by maintaining a "phase table", with each row containing the link vector which represents the phase and a uniquely assigned phase identifier. A runtime phase classification scheme would thus involve recording all the phases that have been previously encountered in the phase table and (at the end of every new interval) comparing the interval's link vector with the link vectors of the previously-recorded phases (which essentially involves searching the phase table). If there is a match, then that interval is classified as belonging to that phase. If a match is not found, it is a new phase and is added to the phase table with the link vector of the interval and a newly assigned unique phase id. This process can be performed dynamically making it a runtime classification scheme.

5.3 Classification Example

Figure 3(a) shows a snapshot of the link vectors (of intervals) during a period of execution of the Wupwise benchmark from the SPEC OMP benchmark suite [2]. In this figure, "count", present in each row, indicates the number of contiguous intervals during which the same link vector repeats. The classification (mapping) of intervals to phases which is based on the link vector similarity can be noted.



	S1	S2	S3	S4
S1	0.983	0.013	0.004	0
S2	1.0	0	0	0
S3	0.5	0	0	0.5
S4	0	0	1.0	0

Fig. 3. (a) shows a snapshot of link vectors of intervals during a period of execution of the Wupwise multi-threaded benchmark and the phases they map to. Mapping is done based on link vector similarity. (b) depicts a Markov based transition graph and the corresponding prediction table. Prediction is made based on the probabilities contained in the prediction table. The transition graph shows the transition probabilities pictorially.

6 Markov Based Prediction

After classifying the intervals into phases as described in the last section, we use a Markov based prediction mechanism to predict the probable link vector of the next interval just before the end of the current interval. Markov based schemes have been used in the past to implement BBV (basic block vector) based phase prediction [19]. This prediction essentially provides the probable link usage information of the next interval. This, in turn can be used to proactively turn off the links which are predicted to be not used and pre-activate links that are predicted to be used. This pre-activation is done just ahead of time so that the activation latency is hidden and the link is ready for use when the next interval begins. If the prediction turns out to be correct, we stand to save power. However, if the prediction turns out to be wrong, there is a two-fold penalty. First, there is the performance penalty in waiting for the correct links to power on which had been turned off because of the misprediction. Secondly, there is also the power penalty in turning off and then turning on additional links. Therefore, prediction accuracies are crucial to the effectiveness of this scheme. We describe two prediction schemes based on the Markov model in the next two subsections.

6.1 Basic Markov Prediction

Markov model is a prediction model used frequently in various domains [18] [16] [24]. A specification of the Markov model contains a set of states and a table, containing the transition probabilities from each state to every other state and itself. With this specification, Markov model can make a prediction about the

next state, given the present state. This prediction is based on the transition probabilities. The transition probabilities are continuously built and updated as and when state transitions happen, and therefore, these transition probabilities, at any instant, are based on the previous transition history. A basic Markov prediction involves considering the present state and searching the transition probabilities from this present state to every state and choosing the transition which has the maximum probability. In our context, a state is nothing but the link vector of an interval. Figure 3(b) illustrates an example of this scheme. It shows Markov based transition probabilities in the form of a graph and a prediction table at the end of the execution chunk shown in Figure 3(a). Each state in Figure 3(b) corresponds to a phase in the phase table of Figure 3(a). The state S1 corresponds to phase 1, S2 to phase 2 and so on. As an example of Markov based prediction, if the current state is S1, the next state is predicted to be S1 again. As another example, if the current state is S2, then the next predicted state is S1. As a simple illustration of the way transition probabilities are continuously updated, if suppose, S4 now transitions to S2, the new transition probabilities from S4 to S1 and S4 to S4 still remain 0, but the transition probability from S4 to S2 changes from 0 to 0.5 and, the transition probability from S4 to S3 reduces from 1 to 0.5.

6.2 Markov Prediction Using a Threshold

This is similar to the basic Markov prediction scheme explained above with one added quality. Instead of making a prediction based on the maximum probability alone, we base the prediction on another parameter called the "threshold". Specifically, we pick the maximum probability prediction and then, check if its probability is greater than or equal to the pre-specified threshold parameter, and if so, we continue as before by choosing the maximum probability next state as our prediction. However, if the maximum probability is less than the specified threshold value, we do not make any prediction. This scheme is intended to weed out predictions which are based on insufficient previous data or are just too close to call. Note that employing a threshold value, in general, decreases the number of mispredictions, as we show later in the results section. For example, in Figure 3(b), if the present state is S3, the previous scheme would have predicted either state S1 or S4 to be the next state. In contrast, the threshold based scheme with a threshold of 0.67 makes no prediction (for the present state S3) since the maximum probability entry in the row is less than the pre-specified threshold value. The threshold value is a configurable parameter and can be set high if very little performance impact is tolerated and can be set low if some performance impact can be tolerated with a possibility of higher energy savings.

7 Implementation

After having described the important concepts we employ, we now present the implementation details of our scheme. Our proposed predictive link power management can be implemented in two possible ways. One way is to consider a

global power manager which manages the power usage of the entire NoC based CMP. Such a global power manager can be implemented as a separate microcontroller to manage chip-wide power usage by controlling the power management of individual cores, as has been previously proposed [14]. It has also been shown that such global power management can be much more efficient and beneficial than local core-level power management schemes [28] [14]. We can extend such a global power manager with a link power management module. Alternately, predictive link management can be implemented using a helper thread which runs parallel to the computation threads and manages the link power. In either case, some hardware support is needed for our scheme to work.

7.1 Required Hardware

Figure 4 shows the hardware details of the link power management module we propose to use. We assume the existence of hardware or software module which can notify us whether a link was used in the previous instruction or not. We now describe the main components and their functionality.

- *Link Vector Register*: This is a 24-bit register that collates the link vector of an instruction interval. This register is reset at the beginning of each interval. After each instruction in the interval, the new link usage information of that previous instruction (in the form of link vector) is ORed into this register. Therefore, at the end of the interval, this register contains the current link vector for the entire interval. Since this is just a single 24-bit register and the operation is relatively simple, its overheads are negligible.
- *Link Vector Table*: This is a 32-entry table that maps each 24-bit link vector entry to a distinct 5-bit phase id. Since this table can contain 32 distinct phases at any given time, phase id is a 5-bit entry. Each row also contains two counters, "correct" and "misprediction", which count the number of correct predictions and the number of mispredictions, respectively. These counters are used to enforce our replacement policy as will be described later. The link vector table is turned on just before the end of each interval to make the prediction. During the rest of the interval, this table is turned to a low-power drowsy mode [10] and hence, the power overhead of maintaining such a table is minimal.
- *Prediction Table*: This is also a 32-row table. This table contains the Markov transition probability information described in Section 6.1. To reiterate, it contains the transition probabilities from each phase (represented by the phase id) contained in the link vector table to all phases contained in the link vector table. This is the table that is used to make the link vector predictions. Like the link vector table, the prediction table is also turned on just before the end of each interval and turned to a drowsy mode [10] otherwise, hence leading to negligible power overhead.

7.2 Functionality

During each instruction interval (which includes 5000 instructions in our experiments), the link usage vector is constructed and stored in the link vector register.

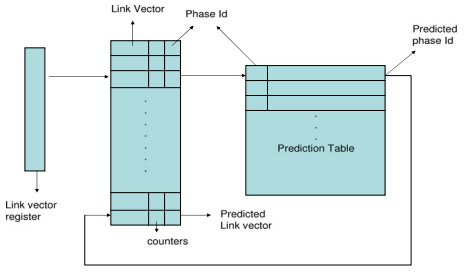


Fig. 4. Hardware needed to implement the Markov based prediction scheme. The main structures needed are the link vector register, the link vector table and the prediction table.

Just before the end of an instruction interval, the link vector table is searched to find if there is an entry which matches the value in the link vector register. If a match is found, this means that this particular link usage phase has been seen previously. The corresponding phase id is taken from the link vector table and provided as an input to the prediction table. The prediction table outputs the predicted phase id of the next interval. This predicted phase id is computed using one of the two Markov based prediction schemes described earlier.

The link vector corresponding to the above predicted phase id is found by searching the link vector table. This link vector represents the predicted link usage during the next interval. Using this, the links which are predicted to be unused are turned off and the links which are predicted to be used but are presently switched off are turned on. We assume hooks to turn-on and turn-off communication links to be present. This whole process is performed just before the beginning of the next interval. How long before the beginning of the next interval a link should be turned on depends on the time needed for the links to turn on. As a result of this, communication links which are predicted to be used are turned on ahead of time considering the link turn-on latency. By doing so, link turn-on latency can be completely hidden and consequently, the potential performance penalty can be avoided. If the prediction is correct, after turning on and turning off appropriate links, the prediction table is updated with the new transition probabilities as described in Section 6.1. On the other hand, if the prediction is wrong, the correct links are turned on, incurring a performance penalty equivalent to the link turn-on latency and also a power penalty. After this, the appropriate counters are updated in the link vector table. If the prediction is correct, the *correct* counter is incremented otherwise the *misprediction* counter is incremented. Therefore, these two counters jointly maintain the information needed to calculate the prediction accuracy of this entry. Later, the transition probabilities are updated as in the case of correct prediction.

7.3 Replacement Policy

The above description is for the case where a match is found in the link vector table. If a match is not found, then this indicates a new phase and needs to be

added to the link vector table and the prediction table. There are two possibilities in this case:

- *Tables are not full*: This is the case where the link vector table and the prediction table are not full. As is obvious, the new phase is added to the link vector table as well as the prediction table. Since this is a new phase, no prediction is made.
- *Tables are full*: If the tables are full, we need to find a *victim* to be evicted to make space for this new phase. The victim is selected based on the prediction accuracies. The entry with the lowest prediction accuracy is selected as the victim and evicted. This is where the correct and the misprediction counters come into picture. Using these counters, the phase entry having the lowest prediction accuracy is identified and evicted. The new phase is now added as described before. Using the above prediction accuracy based eviction scheme, the prediction table entries which have not been predicting well are thrown out. This accuracy-oriented replacement mechanism helps us to keep only the phases with good prior predictions in the tables.

7.4 Discussion

An important requisite of any power saving scheme is two-fold. The first requirement is that of minimal performance penalty. Secondly and importantly, power overhead to maintain the hardware needed for the power saving scheme should be minimal and way lower than the power savings achieved. Since our phase classification and prediction is for instruction intervals (of 5000 instructions) and not for individual instructions, the link-usage prediction, link turn-ons and turn-offs are done *only once every 5000 instructions*. Therefore, the overhead to make the prediction, link reactivation latency and link reactivation penalty are incurred at most only once every 5000 instructions. Also, the amount of storage needed by predictive scheme is minimal. The link vector register is a 24-bit register. The link vector table is a 32-entry table and so is the prediction table. Since the link vector table and the prediction table are turned on fully only at the end of each interval, the power penalty they incur is negligible in practice. We factor all the penalties and overheads in our experiments and as can be seen later, the benefits are still considerable. Also, since the prediction is done just before the beginning of the next interval and since the computation is relatively simple, the performance overhead is also expected to be minimal. In addition, as we show in the sensitivity analysis later, by reducing the prediction table size, overhead is dramatically lowered and still the prediction scheme performs really well. We demonstrate later that, reducing the table sizes from 32 entries to 16 or 8 entries still achieves almost the same energy savings. Misprediction penalty is another concern, but as we demonstrate in the next section, since prediction accuracy is very high, this is not a significant factor either. Nevertheless, our results below include *all* the performance and power overheads incurred by the proposed mechanism. Also, since the technique employed by our scheme is very generic and not tied any particular NoC structure, it is very scalable as NoC sizes increase and is also adaptable across various NoC structures.

8 Evaluation

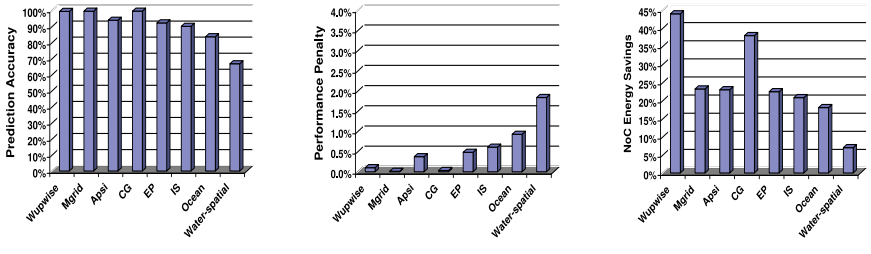
8.1 Setup

As mentioned previously, we use a 4×4 mesh NoC based 16-core CMP in our experiments. We assume a traditional X-Y routing policy in the NoC. The shared L2 cache is 16 banked SNUCA (static non-uniform cache access) architecture with a bank in every node and each bank is 2MB in size. The link power model we use is taken from [34], and in this model, when a link is turned on, it consumes the same power irrespective of whether it is transmitting data or not due to the link signaling methodology. When a link is turned off, we assume it does not consume any power as in [34]. Figure 5 presents the default configuration we use in our experimental setup and in the power analysis. The power values in the table are obtained from [5]. We use Simics [23] which is full-system simulator combined with a module we implemented to simulate a 4×4 mesh. This setup is used to compute link usage, support routing, and evaluate link power management.

We tested our scheme with eight parallel (multi-threaded) applications from three different benchmark suites to find the variation in energy savings across different applications. We selected CG, IS and EP applications from the NAS parallel benchmark suite [1] among which IS and EP are known to have a lot of inter-core communications. From the SPEC OMP benchmark suite [2], we selected Mgrid, Wupwise and Apsi applications. Further, we used the Ocean and Water-spatial applications from the Splash2 benchmark suite [32].

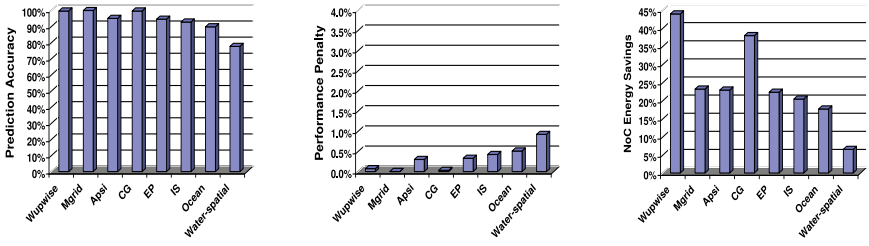
Link frequency	1GHz
Link reactivation delay	1000cycles
Link reactivation energy	36.2nJ
Power of links for an on-chip switch	0.1446W
Process Technology (Interconnect)	0.07μm
Interconnect type	2D mesh NoC
Processor frequency	1GHz
Number of cores	16

Fig. 5. Default system configuration used



(a) Prediction accuracy (b) Performance penalty (c) Energy savings

Fig. 6. Prediction accuracy, performance penalty and the resulting energy savings when the basic Markov prediction scheme is used



(a) Prediction accuracy (b) Performance penalty (c) Energy savings

Fig. 7. Prediction accuracy, performance penalty and the resulting energy savings in the case of Markov prediction using a threshold

8.2 Results

In this section, we present the results for each of the two prediction schemes we presented earlier. For both these schemes, namely, *basic Markov prediction* and *Markov prediction with threshold*, we present the link vector prediction accuracy, the performance penalty incurred and the link energy savings achieved. An important point to be considered for the rest of this paper is that, whenever we present energy savings, we always mean the *effective resulting energy savings*, which is the net energy savings achieved minus the link reactivation energy overhead and the overheads due to additional hardware.

Basic Markov Prediction. Figure 6(a) shows the link vector prediction accuracy achieved by this scheme for various applications. The main observation is the variation in the prediction accuracies across applications. As can be clearly seen, most applications have prediction accuracies of well over 95%, with Wupwise, Mgrid and CG having accuracies over 99%. Compared to this, water-spatial has a slightly lower prediction accuracy, probably due to the relatively shorter execution time, which in turn results in smaller learning phases.

Figure 6(b) shows the performance penalties incurred for different applications, over the case where no link power management is employed. This metric is a reflection of the prediction accuracy. The reason for the observed low penalties is two-fold. The main reason is of course the very high link prediction accuracy. Another reason is the fact that the links that are predicted to be used are turned on ahead of time so that the turn-on latency is hidden and the links are up by the time they are going to be used. The main triumph card of our scheme is the extremely low performance penalties which virtually leaves the original performance unaltered. This is in contrast to other hardware schemes which in many cases incur penalties as high as 12%, as mentioned in [20]. In contrast, our scheme results in penalties below 0.5% in most cases except for water-spatial application, which incurs a penalty of 1.5%. As we demonstrate later, the penalties can be further reduced to being almost negligible.

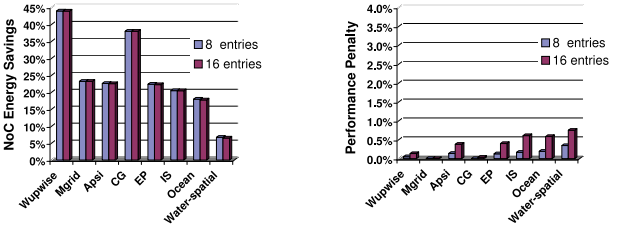
Finally, Figure 6(c) shows the link energy savings achieved by this scheme and as can be seen, Wupwise and CG achieve savings as high as 44% in communication energy. While we present only the NoC energy savings in detail here, our experiments showed that, for the benchmarks we tested, NoC energy consumption constitutes nearly 30% of the total on-chip energy consumption (on-chip energy consumption includes energy consumed by the processing cores, NoC, all cache accesses and other on-chip transactions). This clearly indicates that, NoC energy consumption forms a major component of the total on-chip energy consumption and any significant savings in NoC energy consumption is bound to translate into significant savings in the total on-chip energy consumption.

Markov Prediction Using a Threshold. Figure 7(a), Figure 7(b) and Figure 7(c) show the prediction accuracy, performance penalty and the energy savings, respectively, resulting from this scheme with a pre-specified threshold of 0.5. Later, we also present the results with a different threshold value. Again, the key thing to note is the fact that the performance penalty is further reduced as can be seen in Figure 7(b) and yet the energy savings remain almost the same as in the basic Markov prediction scheme. Hence, incorporating a pre-specified threshold results in further fine tuning of the performance penalties. This happens since the threshold parameter filters out predictions which do not have a good prediction history. Employing this scheme results in performance penalty of less than 1% in all cases and less than 0.5% in all but one application.

8.3 Sensitivity Experiments

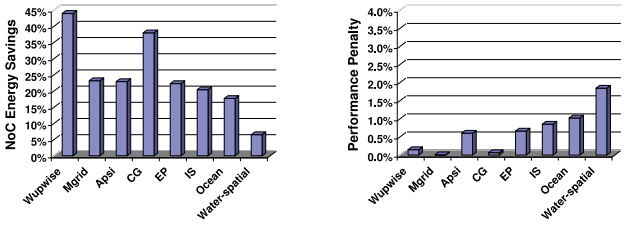
In this section, we study the sensitivity of our scheme to various parameters. We alter aspects such as the prediction table size and present the variation in the results. We first reduce the sizes of the link vector table and the prediction table. We reduce the table sizes from 32 entries to 16 entries and 8 entries. Since these tables are the major storage structures used in our prediction scheme, we intend to reduce the sizes in order to further mitigate the power and computation overheads. We use the Markov prediction scheme using a threshold of 0.5 as presented previously. Figure 8 shows the new energy saving and the performance penalty when the table sizes are reduced. As can be clearly seen, the energy gains remain almost the same and surprisingly, there is also a slight reduction in the performance penalty. This reduction is due to our prediction accuracy based replacement policy, which throws out entries with lower prediction accuracies due to the reduced table size, and in the process also reduces the number of mispredictions we experience. These results show clearly that our scheme works beneficially even when the hardware table sizes are as low as just 8 entries.

The two potential factors which can limit the gains achieved by our scheme are the link reactivation energy and the link reactivation latency. In order to find the dependence of the benefits achieved by our scheme on these factors, we doubled the link reactivation latency and the link reactivation energy values. We then tested the Markov prediction scheme using a threshold value of 0.5, table size of 32 and with the new, doubled link reactivation latency and link



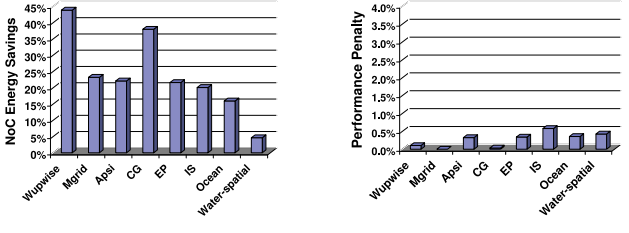
(a) Energy savings (b) Performance penalty

Fig. 8. Energy saving and the performance penalty values when the table size is reduced to 16 and 8 entries. The prediction scheme used here is the Markov prediction using a threshold of 0.5.



(a) Energy savings (b) Performance penalty

Fig. 9. Energy savings and the performance penalty when the link reactivation energy and the link reactivation latency values are doubled. The prediction scheme used here is the Markov prediction using a threshold of 0.5.



(a) Energy savings (b) Performance penalty

Fig. 10. Energy savings and the performance penalty achieved in the case of Markov prediction using a threshold when the threshold value is set to 0.8

reactivation energy values. The corresponding results can be seen in Figure 9. There is a slight increase in the performance penalty, but even with the increase it is well under 1% for all benchmarks except water-spatial, which has a penalty of just over 1.5%. The reduction in energy savings is negligible as can be seen in Figure 9. This clearly indicates that, even with high reactivation penalties, our

scheme performs well and this in turn implies that, the energy saving achieved is not overly sensitive to reactivation penalties.

In the Markov prediction scheme using a threshold, we had earlier used a pre-specified threshold of 0.5. We increased the threshold to 0.8 to check for further reduction in performance penalty. We repeat the experiments with Markov prediction scheme using a threshold value of 0.8 this time around. Figure 10 shows the new results. There is indeed a reduction in performance penalty but the energy savings achieved still remains largely same. This reduction in the performance penalty is due to further fine tuning by the higher threshold value. The higher threshold prevents predictions which do not necessarily have very high probabilities and hence further decreases mispredictions.

9 Concluding Remarks

The goal of this paper is to propose a runtime, proactive scheme for link energy reduction in NoC based CMPs. To that end, we have made the following contributions. First, we have proposed a link usage based dynamic program phase characterization and a technique to use this phase characterization to implement a prediction based pro-active link power management scheme. Second, we have employed this scheme and conducted experiments with various parallel benchmarks and found that our scheme achieves up to 44% in energy savings, and an average saving of about 23.5%. Third, we have found that the most important advantage of our scheme is the remarkably low misprediction rate and hence the low performance penalty which in most cases is below 0.5%.

References

1. http://www.nas.nasa.gov/resources/software/npb.html
2. http://www.spec.org/omp/
3. Cell broadband engine - white paper. IBM (2006)
4. Benini, L., Micheli, G.D.: Powering networks on chips: Energy-efficient and reliable interconnect design for socs. In: Proc. ISSS (2001)
5. Chen, X., Peh, L.-S.: Leakage power modeling and optimization in interconnection networks. In: Proc. ISLPED (2003)
6. Dally, W.J., Towles, B.: Principles and Practices of Interconnection Networks. Morgan Kaufmann, San Francisco (2004)
7. Dhodapkar, A.S., Smith, J.E.: Managing multi-configurable hardware via dynamic working set analysis. In: Proc. ISCA (2002)
8. Dhodapkar, A.S., Smith, J.E.: Comparing program phase detection techniques. In: Proc. MICRO (2003)
9. Duato, J., et al.: Interconnection Networks: An Engineering Approach. IEEE CS Press, Los Alamitos (1997)
10. Flautner, K., et al.: Drowsy caches: Simple techniques for reducing leakage power. In: Proc. ISCA (2002)
11. Galles, M.: Spider: A high-speed network interconnect. IEEE Micro. 17(1), 34–39 (1997)

12. Hetherington, R.: The UltraSparc T1 processor. SUN (2005)
13. Huang, M.C., et al.: Positional adaptation of processors: Application to energy reduction. In: Proc. ISCA (2003)
14. Isci, C., et al.: An analysis of efficient multi-core global power managment policies: Maximizing peformance for a given power budget. In: Proc. MICRO (2006)
15. Isci, C., Martonosi, M.: Phase characterization for power: Evaluating control flow based and event counter based techniques. In: Proc. HPCA (2006)
16. Joseph, D., Grunwald, D.: Prefetching using markov predictors. In: Proc. ISCA (1997)
17. Kim, C., et al.: Nonuniform cache architecture for wire-delay dominated network-on-chip caches. In: IEEE Micro.: Micro's Top Picks from Computer Architecture Conferences (2003)
18. Latouche, G., Ramaswami, V.: Introduction to matrix analytic methods in stochastic modeling. In: ASA SIAM, PH Distributions (1999)
19. Lau, J., et al.: Transition phase classification and prediction. In: Proc. HPCA (2005)
20. Li, F., et al.: Compiler-directed proactive power management for networks. In: Proc. CASES (2005)
21. Li, F., et al.: Profile-driven energy reduction in network-on-chips. In: Proc. PLDI (2007)
22. Li, J., Martinez, J.F.: Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In: Proc. HPCA (2006)
23. Magnusson, P.S., et al.: Simics: A full system simulation platform. Computer 35(2), 50–58 (2002)
24. Oly, J., Reed, D.A.: Markov model prediction of i/o requests for scientific applications. In: Proc. ICS (2002)
25. Perelman, E., et al.: Detecting phases in parallel applications on shared memory architectures. In: IPDPS (2006)
26. Ramanathan, R.: Intel multi-core processors: Making the move to quad-core and beyond. Intel White paper, Intel Corporation (2006)
27. Shang, L., et al.: Dynamic voltage scaling with links for power optimization of interconnection networks. In: Proc. HPCA (2003)
28. Sharkey, J., et al.: Evaluating design tradeoffs in on-chip power management for cmps. In: Proc. ISLPED (2007)
29. Sherwood, T., et al.: Discovering and exploiting program phases. In: IEEE Micro: Micro's Top Picks from Computer Architecture Conferences (December 2003)
30. Shin, D., Kim, J.: Power-aware communication optimization for network-on-chips with voltage scalable links. In: Proc. CODES+ISSS (2004)
31. Simunic, T., Boyd, S.: Managing power consumption in networks on chips. In: Proc. DATE (2002)
32. Singh, J.P., Weber, W.-D., Gupta, A.: Splash: Stanford parallel applications for shared-memory. Computer Architecture News 20(1), 5–44
33. Soteriou, V., Peh, L.-S.: Dynamic power management for power optimization of interconnecttion networks using on/off links. In: Proc. HOT-I (2003)
34. Soteriou, V., Peh, L.-S.: Design space exploration of power-aware on/off interconnection networks. In: Proc. ICCD (2004)
35. Timothy, G.H., Mattson, G.: An overview of the Intel TFLOPS Supercomputer. Intel. Technology Journal (1998)

Mapping and Synchronizing Streaming Applications on Cell Processors

Maik Nijhuis[1], Herbert Bos[1], Henri E. Bal[1], and Cédric Augonnet[2]

[1] Vrije Universiteit, Amsterdam, The Netherlands
{maik,herbertb,bal}@cs.vu.nl
[2] INRIA - LaBRI, Université Bordeaux 1, France
cedric.augonnet@inria.fr

Abstract. Developing streaming applications on heterogenous multi-processor architectures like the Cell is difficult. Currently, application developers need to know about hardware details to deal with issues like scheduling, memory management and communication/synchronization. Worse, with multiple alternatives for communication available, developers spend significant time picking the most appropriate one. A poor choice often results in bad performance. With Cell-Space, we shield users from hardware details without compromising performance. Its runtime is based on an evaluation of the different communication primitives. In Cell-Space, developers specify a streaming application as a data flow graph of interacting components. Both task- and data-parallelism are easily expressed and advanced features such as dynamic reconfiguration are fully supported. Beneath a simple interface we include a slew of optimizations not present in other Cell run time environments. We demonstrate the impact of these optimizations and show that Cell-Space applications can efficiently exploit the resources offered by the Cell.

1 Introduction

Streaming applications underly a variety of application domains including audio/video, networking, and processing of extremely large data sets. Moreover, much of the consumer electronics industry hinges on streaming. As streaming data is only valid for a limited time, we are forced to process the data in line. On the one hand, keeping up with growing data rates and processing demands is challenging even on modern processors. On the other hand, streaming applications exhibit much potential parallelism. For instance, they may (a) process data in a pipeline of stages, or (b) spread it in a SIMD fashion over a number of identical functional units, or (c) divide a stream in sub-streams (e.g., audio and video) and process each of them differently. In fact, we would probably like to use complex combinations of data and task parallelism.

Heterogeneous multi-core processors like the Cell [1] appear to be a perfect match for these applications. After all, we can run the control part of an application on a Cell's general purpose Power core (the Power Processing element, or PPE) and push all data processing to its specialized SIMD RISC cores (known as Synergistic Processing Elements, or SPEs). The SPEs' 128-bit SIMD organization and fast, private memory make them ideal for stream processing. With eight such cores on a die, the Cell is

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 216–230, 2009.

one of the most powerful processors currently available. Moreover, vendors often pack multiple Cells in a single blade server. As a result, the Cell is used in machines ranging from game consoles to supercomputers [2].

However, in practice, complex communication and scheduling requirements make streaming applications challenging even on single core architectures. Heterogeneous multi-cores like the Cell add yet another difficulty layer. The question is then: why is it so difficult to map streaming applications on the Cell? In our experience, the problem is caused primarily by the need to find efficient solutions to the following issues:

1. Scheduling and resource utilization (load-balancing);
2. Memory management with distributed memory and inter-core data transfers;
3. Communication and synchronization (efficient notification messages).

Due to lack of high-level programming support, all of these issues have to be handled explicitly by the application programmer, which implies that the programmer has to worry about low-level, architecture-specific details. Making the wrong implementation decision results in poor performance, as all of the above issues are crucial for efficiency. Asking application developers to worry about low-level hardware-specific optimizations borders on the unreasonable and leads to poor portability.

To make matters worse, it is very difficult for the programmer to make an informed decision about *which* mechanism to use for low-level implementation issues like how to handle data transfers and synchronizations. The Cell offers a range of options but it is unclear which are most suitable for what purposes. To achieve good performance, programmers are forced to consider a host of design alternatives. For instance, they must worry about: the pros and cons of interrupts versus DMA, the optimal size of code executing on SPEs, how to split up their applications in components, how to schedule jobs on SPEs and how to decide under what circumstances which parts of the application should be scheduled on what SPEs, etc. These issues are in addition to developing multi-buffering schemes which overlap processing, efficient data transfer, and dealing with more than one Cell processor. The problem gets even uglier when application configurations change at runtime (e.g., a TV that adds or removes picture-in-pictures, PiPs, because the user presses a button).

In our opinion such demands on application developers are undesirable, unreasonable, and unnecessary. Experience in the related field of network processors has shown that it also poses a real threat to the success of the architecture.

Contributions. In this paper, we describe Cell-Space, a framework for developing streaming applications on the Cell. In Cell-Space, developers use a high-level coordination language for constructing an application, in the form of a data flow graph, out of components in a component library (see Fig. 1). The graph is first translated to an intermediate XML-based language and, after various optimizations, compiled to C. When the application is deployed, a runtime system on the PPE schedules the runnable components in a load-balancing fashion over the available Cell processors. The components in turn use a runtime library to run jobs on the SPEs. Cell-Space handles all complex Cell-specific issues, including scheduling, memory management, data transfers, communication and notification.

Our main contributions are:

1. A high-level model which helps building streaming applications from components;
2. A runtime system which schedules components over the available cores in a load-balancing fashion with support for reconfigurability;
3. A runtime library which encapsulates all of the previously mentioned complexity such as multi buffering, synchronization mechanism, code size adaptation, etc.

The runtime library itself is based on an additional contribution, which is that we are the first to report on the relative merits of the various communication and notification mechanisms of the Cell. We evaluated these mechanisms and encapsulated the most optimal mechanisms behind a convenient API. Finally, we evaluate the system not just by means of synthetic micro-benchmarks, but also by way of real applications.

To the best of our knowledge, we are the first to design and implement such an integrated approach for developing for Cell-like architectures which supports reconfigurability. Moreover, even the various parts of Cell-Space offer advantages over competing projects. For instance, the Cell runtime library in Cell-Space is considerably friendlier than related projects like ALF and Charm++ [3,4]. At the same time, it provides a slew of optimizations not present in its counterparts. Other run time systems exist that schedule data flow graphs and balance the load over available processors, and some, like StreamIt [5], even support the Cell processor. However, none of them seem to have support for reconfigurability.

Paper outline. The focus of this paper will be on the components responsible for ensuring good performance and utilization: the runtime library and, to a lesser extent, the runtime system. Specifically, we will look at the way applications are structured conceptually, how they are scheduled by the runtime, and how they use the runtime library for efficient communication and synchronization. The front-end and intermediate representation are discussed in more detail in [6].

The remainder of this paper is organized as follows. First, we give a high-level overview of Cell-Space in Sect. 2. Section 3 presents its implementation as well as the various optimizations we perform. The impact of these optimizations and the overhead of Cell-Space is evaluated in Sect. 4. Finally, Sect. 5 presents related work and Sect. 6 concludes the paper.

2 The Cell-Space Architecture

Figure 1a sketches a high-level overview of the Cell-Space development model. Developers use a high-level front end for constructing data flow applications from a library of Cell-Space components. The model has no implicit preference for any particular front end. Obvious candidates for the front-end are GUI-based environments in which applications are constructed by clicking together components graphically. The only requirement is that the front-end generates programs in the Cell-Space intermediate language which presents an application as an XML description of a data flow graph consisting of connections and (possibly nested) components.

The XML-based intermediate language is not just a convenient target for the front-end and an easy-to-parse input for the back-end. It also serves as source and target

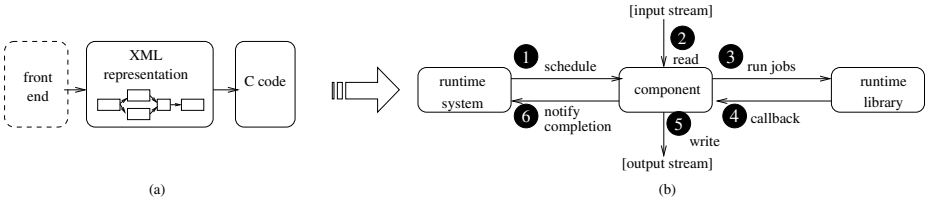


Fig. 1. High-level Cell-Space model

for various XSL transformations and optimizations, such as loop unrolling and conversion of data parallelism to task parallelism where possible. As an aside, in practice the language is quite readable for end users (sufficiently so that thus far we have not yet bothered to write a mature front end, writing all of our applications directly in XML).

After the XML transformations, the intermediate representation is compiled to C and eventually linked to the Space runtime system (*Space-RTS*, see Fig. 1b). Space-RTS is responsible for scheduling the components for which all inputs are available on the various processors in the system. The components themselves read data from their input streams. Components may use the Cell runtime library (*Cell-RTL*) for running jobs on the SPEs. Cell-RTL notifies the components using a call back mechanism when the jobs have completed. When the component itself completes, it notifies Space-RTS.

The glue between Space-RTS and Cell-RTL is formed by so-called Cell components. Towards the Cell-Space framework, Cell components act like normal components. At the PPE, they adhere to the standard component interface, and use the primitives provided by the run time system for streaming and event communication. Internally, they use the Cell-RTL library for offloading their computations to the SPEs. The basic interaction in Cell-Space is illustrated in Fig. 1. Space-RTS schedules a Cell component (①) which reads its input streams (②) using Space-RTS's streaming communication interface, and uses Cell-RTL to run jobs (③) on the SPEs to process this data. When jobs complete, Cell-RTL invokes a call back function (④) in the component, which then writes the output data (⑤) from the SPEs to its output streams. Finally, the Cell component notifies Space-RTS (⑥) that it has finished executing and thereby allows Space-RTS to schedule its successors in the data flow graph.

Space-RTS – a runtime system for streaming applications. The Space-RTS runtime system abstracts the programmer from difficulties of the parallel architecture, such as load balancing, synchronization, and communication between the main components of the streaming application. Space-RTS organizes the components in a data flow graph that corresponds to the graph in the intermediate XML representation and provides both streaming and event communication primitives to the components. Using central and distributed queues, Space-RTS dynamically balances the load over the available processors. In addition, it fully supports advanced constructs, such as end-user event handling and dynamic reconfiguration. Due to its modular design, Space-RTS can easily be extended to support even more advanced applications in the future.

Reconfigurability is sparked by events that are either caused by user actions (e.g., a button to add a picture-in-picture), or generated by other components. It is a complex

operation, that requires removing, adding, and changing components in a running data flow graph. We handle asynchronous events by buffering them in event queues, which are periodically emptied by a manager component.

Reconfiguration of the data flow graph is supported by the general component interface which supports combining several components in groups. Components can be dynamically created, destroyed, grouped, and connected at run time. To avoid race conditions, the application parts that are reconfigured are made idle before reconfiguring. As we control the activity within the data flow graph, this operation is always possible.

Dynamic load balancing greatly assists dynamic reconfiguration. With static load balancing, the compiler generates a schedule for each possible configuration. The number of schedules grows exponentionally with the number of configuration options. Our approach is more elegant as it does not require these schedules. Furthermore, since a schedule captures the full application, the application is fully halted at each reconfiguration. In our approach, the parts that are not reconfigured keep running at full speed.

Cell-RTL – a runtime library for running jobs on SPEs. Cell-RTL is the run time library that facilitates Cell programming by providing a simple interface for using the SPEs. Although Cell-RTL has been developed specifically for the Cell, its simple interface and many optimizations may be applied to similar MPSoC architectures as well. The API of Cell-RTL is based on offloading *jobs* to the SPEs. A job is a self-contained application part that performs some computation at an SPE on input data and produces output data. In streaming applications, these computations consist of kernels and filters, to which Cell-RTL passes the local addresses of the input and output data at the SPE. All jobs are represented by a higher-level Cell-Space component in a $1 : n$ relationship. Thus, Cell-RTL is responsible for low-level synchronization and communication to the jobs on the SPEs, while Space-RTS handles the higher-level synchronization between Cell-Space components. Functionality-wise, Cell-RTL relieves the programmer of the following difficult tasks:

- SPE management. Cell-RTL performs all SPE management tasks, including initialization, memory management, scheduling, and exception handling.
- Load balancing. Cell-RTL dynamically assigns jobs to the SPEs, based on their availability. When all SPEs are busy, Cell-RTL internally queues new jobs. When an SPE completes a job, Cell-RTL sends a job from this queue to the SPE.
- Communication. Cell-RTL performs all communication with the SPEs, which includes transferring input and output data between main memory and SPE local memory, sending jobs to the SPEs, and sending notification messages to the PPE.
- Synchronization. Because the SPEs run asynchronously to the PPE, Cell-RTL synchronizes the PPE and the SPEs regularly.

In addition, we will now show that Cell-RTL implements a slew of optimizations like multi-buffering, job chaining, and efficient communication.

3 Implementation

Conceptually, the responsibilities of Cell-RTL and Space-RTS mentioned in the previous section are straightforward. For instance, Space-RTS should track dependencies

and schedule components when all required inputs are available. Similarly, a Cell-RTL job has input and output buffers. Cell-RTL needs to DMA the input data to the SPE's local memory and relay results have back to the components running on the PPE. However, efficiently implementing these communication and synchronization mechanisms requires a lot of knowledge about the low-level details of the Cell processor.

For Cell-Space, we systematically analyzed and evaluated various alternatives to arrive at a highly efficient runtime. As mentioned earlier, by hiding the details behind Space-RTS and Cell-RTL, Cell-Space shields developers from the complex implementation details and tradeoffs. Nevertheless, we believe that both the issues we considered and our findings are essential for anyone developing applications or runtimes for Cell-like processors. For this reason, we now discuss the most important results.

3.1 Asynchronous Notification

The nature of synchronization on processors like the Cell is such that upon completion of a computation cores need to notify other cores. This is typically done by means of a small identifier, such as an integer. In Cell-Space, when a job is complete, the SPE sends a single 32 bit notification message to the PPE. The Cell processor has two special-purpose mechanisms which are intended for transferring these messages, namely interrupts and outbound mailboxes. The default DMA communication mechanism can also be used. Figure 2 shows the three approaches, which are described below.

For evaluation purposes, we created three versions of Cell-RTL using interrupts, outbound mailboxes, and DMA, respectively. We describe each of them below. The full evaluation is described in Sect. 4, but as a preliminary result we mention that, surprisingly perhaps, DMA outperforms both special-purpose mechanisms. Cell-RTL therefore uses DMA for notifications by default.

Interrupts. Using an interrupt outbound mailbox, an SPE can trigger an interrupt at the PPE. The PPE then reads the 32 bit mailbox content. The PPE runs a separate thread that is woken up at every SPE interrupt. It is similar to softirqs in the Linux kernel [7], as this thread does not run in strict interrupt mode.

Although this approach is relatively straightforward, it has two main disadvantages. First, interrupts are costly because they are handled by the OS [8]. When an interrupt arrives, the interrupt handler is invoked. Also, the OS makes a context switch to the interrupt handling thread. Second, interrupt communication does not scale because there is only one interrupt mailbox for the entire Cell processor, which is shared among all SPEs. When one SPE has sent an interrupt message, the other SPEs have to wait until the PPE has processed it before they can send another interrupt message.

Mailbox. Besides the interrupt outbound mailbox, there is one normal outbound mailbox per SPE, which also holds a 32 bit message. When the SPE uses this mailbox to send a message to the PPE, no interrupt is generated. Therefore the PPE needs to poll the mailbox periodically to see if a new message is available.

This approach has two different disadvantages. First, polling the outbound mailbox for new messages incurs some overhead because it requires a system call to the OS. Moreover, the mailbox resides on the SPE, and not on the PPE. For each poll, the

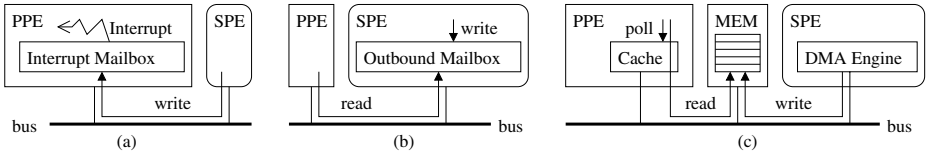


Fig. 2. Notification approaches: (a) Interrupts, (b) Mailbox, (c) DMA

PPE makes a transaction over the internal element interconnect bus (EIB) to access the mailbox at that SPE. Second, each SPE has only one mailbox slot. If an SPE wants to acknowledge multiple jobs, it has to wait until the PPE has read the previous message.

In Cell-RTL, the PPE automatically polls all SPEs whenever the application submits a job. The polling rate is therefore automatically adjusted to the job submission rate. The application can also poll or wait explicitly in case it does not submit new jobs.

DMA. The SPEs can also use DMA for sending 32 bit messages to the PPE. Cell-RTL assigns a special DMA region for each SPE to which it writes job acknowledgment messages. These regions contain a fixed number of message slots, which are used in cyclic order.

With the DMA approach, polling is done by simply reading memory. Similarly to NAPI [9], multiple slots can easily be polled at once, draining all pending notifications in one go. Unsuccessful polls are usually done using the PPE's internal cache, without accessing the internal interconnect bus. By enforcing a maximum number of outstanding jobs per SPE, the PPE ensures that the acknowledgment slot for a certain job is always available. The statically allocated circular buffer is an efficient structure as it provides a lock-free producer-consumer channel and incurs no runtime allocation. Similar constructs are used in various other systems [9,10,11].

Unlike the other approaches, the SPE does not have to wait for the availability of a mailbox slot when acknowledging a job. It only has to schedule an asynchronous DMA request. Only when its DMA engine is fully occupied, it has to wait. However, this condition is unlikely to occur, as the SPE DMA engine can queue up to 16 DMA transfers. When the queue has an available slot, the DMA request is handed over to the DMA engine and the SPE immediately continues processing other jobs.

Similarly to the mailbox approach, Cell-RTL automatically adjusts the polling rate to the job submission rate by polling all notification buffers whenever a job is submitted. Again, the application can poll or wait explicitly when it does not submit new jobs.

3.2 Multi-buffering and Chaining

Cell-RTL allows multiple pending jobs at a single SPE for overlapping communication and computation using multi-buffering. Before executing a job, Cell-RTL initiates asynchronous DMA transfers to fetch the input data for other pending jobs. It also sends the output data using asynchronous DMA. While executing a job, Cell-RTL thus transfers both the input data for subsequent jobs as well as the output data from previous jobs.

Spawning small jobs on the SPEs is expensive as each job incurs notification and transfer overheads. Cell-RTL therefore allows for the combination of several jobs into a *job chain*, which is a list of jobs that is scheduled as one entity. Job chaining reduces overhead as many costs, such as those of synchronization or memory allocation, are incurred once per chain, rather than once per job. Job chaining also allows data transfer and memory re-use between jobs in a chain, which eliminates data transfers to and from main memory when the chain contains producer-consumer jobs.

3.3 Starting Jobs: Data Transfer and Asynchronous Execution

The protocol for data transfer from the Space-RTS streams to Cell-RTL jobs minimizes copying (see Fig. 3). When the component reads or writes a stream, it receives the address in memory of a buffer in the stream from Space-RTS. The component in turn enters this address in the job description of the SPE jobs. Cell-RTL then submits the addresses to the SPEs. The PPE performs no data copying at all, it only copies addresses. The SPE executing the job still transfers the data to its local memory using DMA as it can not directly access main memory.

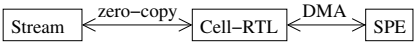


Fig. 3. Zero-copy protocol

A normal Cell-Space component, which only uses the PPE, returns control to Cell-Space after it has finished running and all output data streams have been written. When a Cell-RTL component has submitted its jobs to Cell-RTL, it does not wait until Cell-RTL has finished running these jobs. Instead, it returns control to Cell-Space which allows Cell-Space to run other components, including other Cell-RTL components. Running Cell-RTL components asynchronously has important advantages, First, there is no context switching or thread management overhead as this approach requires only a single thread. Second, the number of active Cell-RTL components is unlimited. Cell-RTL always accepts new jobs and maintains an internal queue of pending jobs at which jobs are put if all SPEs are busy. Third, the main processor and the SPEs are optimally used. The main processor is always available for normal components as Cell-RTL components do not wait. The SPEs are optimally used as Cell-RTL receives all jobs as soon as Space-RTS schedules the corresponding Cell-RTL component.

4 Evaluation

We evaluate Cell-Space using two first-generation Cell processors. Each Cell processor has eight available SPEs. As Cell-Space distributes load over both processors, we effectively have 16 SPEs and two PPEs. Both the PPEs and the SPEs run at 3.2 GHz. We measure execution time using the built-in decrementer register, which ticks every 120 clock cycles. We repeated the experiments 11 times, after which we took the average.

We first measure the overhead of Cell-RTL using a synthetic SPE benchmark function, which executes a fixed number of cycles on the SPE when Cell-RTL invokes it. The total number of cycles spent in all invocations of this function on all SPEs is divided by the number of SPEs, which results in the average effective execution time (AEET).

On the PPE, we measure the total execution time (TET) on the application. The measurement starts just before the application submits the first job to Cell-RTL and

ends when all jobs have finished. It does not include initialization, e.g., set up cost, and finalization, e.g., shutdown cost and printing the result of the measurement.

The relative and absolute overhead of using Cell-RTL are derived from the AEET and the TET. The absolute overhead is the TET minus the AEET, which is zero in the optimal case. The relative overhead is the absolute overhead divided by the AEET, and is given as a percentage. These figures thus include the cost of DMA transfers and the overhead of Cell-RTL. Cell-RTL has overhead both on the PPE side in the interaction with the application and the SPEs, and in the management code that runs on the SPEs.

4.1 SPE Functions

We determine typical computation to communication ratios for SPE jobs by analyzing several SPE functions that are used in real applications. We have created stand alone SPE programs that run the specified function and measure the number of cycles used. The total input and output data size is set to one quarter of the total SPE local memory, which allows multi-buffering when these functions are used in real applications.

All functions in the SPE function library perform image processing. The input and output pixels for these functions are represented using one byte per color component. Using SIMD optimizations, they perform the following operations:

- The maximum function takes the maximum of corresponding values in its input buffers. It processes 64 pixels of one byte at once.
- Yuv2rgb converts color pixel data in YUV format to RGB format. It converts 16 pixels at once.
- IDCT performs an Inverse Discrete Cosine Transform of 8x8 pixel blocks.
- The convolution functions apply a 5x1 or non-separated 3x3 Gaussian blurring kernel to the input image. The input image contains a border of 8 pixels on the left and right sides, which is not present in the output image. With the 3x3 kernel, additional borders of 1 pixel are added at the top and bottom of the input image.
- The geometric transformer applies an affine transformation to the input image. Its input is a block of pixels from the input image. Its output is the corresponding block of pixels in the output image. Its arguments include the transformation matrix, and the position of both the input and output block in the full images.

Table 1 lists the total number of bytes transferred for each kernel along with a measured decrementer tick count. From these figures, we compute the number of cycles per transferred byte. This value differs by an order of magnitude. We take this into account when evaluating Cell-RTL because the computation to communication ratio has a significant impact on the overhead of Cell-RTL.

4.2 Notification Approaches

We evaluate the different notification approaches modes described in Sect. 3.1 using a synthetic benchmark application. The benchmark uses the three approaches with 16384 jobs with 32400 input bytes and 32400 output bytes. The number of decrementer ticks per job is 540 or 5400 which leads to 1 or 10 clock cycles per transferred byte, respectively. This ratio complies with the results from Sect. 4.1.

Table 1. Computation to communication ratios for various kernels

Kernel	Bytes	Ticks	Cycles/B
Maximum	61952	346	0,670
yuv2rgb	61444	379	0,740
IDCT 8x8	51208	643	1,507
5x1 convolution	66740	4032	7,250
3x3 convolution	64238	4560	8,518
geom. transform	51248	8440	19,763

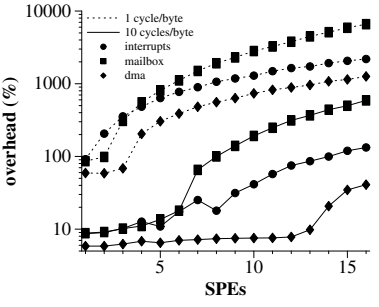


Fig. 4. Notification approach comparison

Figure 4 shows the results of running these benchmarks on 1 to 16 SPEs Note the logarithmic scale of the y axis. The figure shows three important results:

1. Computationally dense kernels should be preferred as we observed that a high computation to communication ratio implies low overhead. Overhead is greatly reduced by increasing the number of cycles per byte, for example, by combining multiple operations that would otherwise be separate jobs.
2. The overhead increases with the number of SPEs, because there is more resource contention. Also, load imbalance is more likely with more SPEs.
3. There are considerable differences between the various acknowledgment modes. The DMA mode clearly outperforms the interrupt mode, which in turn outperforms the mailbox mode. We conclude that the special interrupt and mailbox communication primitives in the Cell do not provide any added value over the default DMA communication primitive. We have therefore chosen DMA as the notification mechanism within Cell-RTL. In our next experiments, we will only use DMA.

4.3 Multi-buffering

Cell-RTL performs multi-buffering on both input and output data. We evaluate this optimization by varying the maximum number of jobs that Cell-RTL concurrently processes on a single SPE. With one job per SPE, multi buffering is not possible. The multi-buffering opportunities increase with the number jobs per SPE, however, when jobs have 64kB of data, an SPE can only hold the data of three jobs because it has limited memory.

Figure 5 shows that for small jobs with 1 cycle/byte, running multiple jobs per SPE increases overhead. Because the memory bus is overloaded, processing the extra job slots increases overhead. With 10 cycles/byte, multi-buffering decreases overhead from 13 % to 7.5 % with 13 or less SPEs. Beyond 13 SPEs, Cell-RTL's aggressive multi-buffering strategy overloads the bus with DMA transfers and overhead increases again. Fortunately, job chaining overcomes this problem, as explained below. Since we mainly run large jobs, we use 2 jobs per SPE in our other experiments because this setting yields the best results for large jobs.

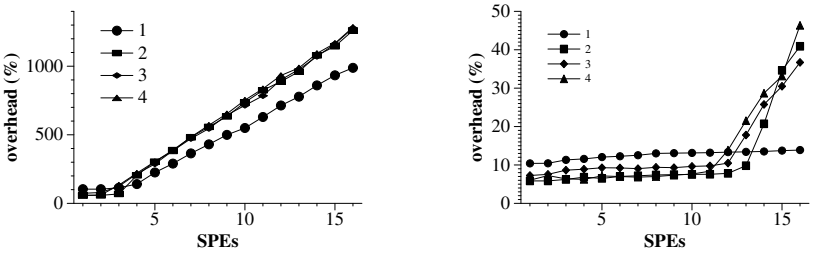


Fig. 5. Evaluation of multi buffering: 1 to 4 pending jobs per SPE

4.4 Job Chaining

For evaluating the impact of using job chaining and persistent data, we run the synthetic benchmark application using three modes:

- Basic. No optimizations are performed.
- Chaining. Instead of scheduling 16384 individual jobs, we schedule 4096 chains with 4 jobs each, or 1024 job chains with 16 jobs each.
- Persistent data. We perform chaining and add persistent input buffers of 12960 or 32400 bytes, which is 20% or 50% of the total data, respectively. We only transfer the persistent buffer with the first job of each chain, however, it remains on the SPE during the execution of the chain. All jobs in the chain can therefore access and even modify this buffer. We reduce the normal input and output buffers by 6480 or 32400 bytes, respectively. The total data size therefore remains equal.

Figure 6 shows the results using 8 and 16 SPEs. On other numbers of SPEs we experienced similar results. Chaining alone effectively reduces overhead: With chaining and 10 cycles/byte, the overhead peak at 16 SPEs (40 %) is completely gone. With 10 cycles/byte, the overhead of Cell-RTL becomes less than 6%. Although using persistent data reduces overhead, the overhead reduction with 20% persistent data is limited because adding an extra persistent buffer increases buffer management overhead. With 50% persistent data, the overhead reduction is more prevalent.

4.5 Application Performance

We now show the effectiveness of Cell-Space using several challenging streaming applications. All applications have an output component which normally stores the application output in a file or displays it on the screen. Using these components, we have verified the correctness of the application. Since we are interested in the performance of Cell-Space, and the performance of external output devices can be a bottleneck, we replaced the output component by a dummy when performing benchmarks. We examine the following applications:

- The JPiP application decodes 16 motion JPEG (MJPEG) streams with a resolution of 1280x720 and combines them into a 4x4 tiled display. An advanced SPE kernel

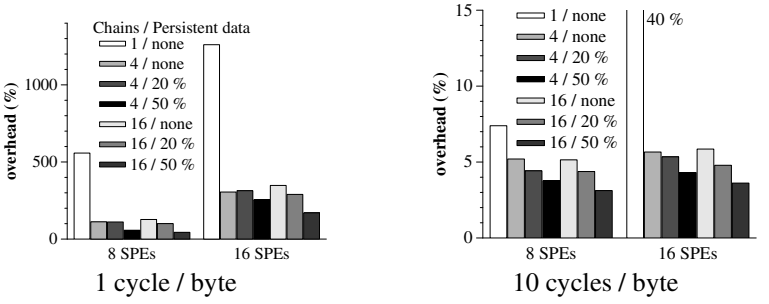


Fig. 6. Evaluation of chaining and persistent data

performs JPEG decompression and down scaling using the IDCT kernel, as mentioned in Sect. 4.1. When all 16 images have been decoded, the PPE blends them into a 4x4 tiled display, which it sends to the output.

- The Edge-Rot application performs edge detection by rotating the input images over 36 different angles. The SPEs perform the rotations using the geometric transformation kernel. For each rotation angle, the PPE generates a border around the input image which is needed for the following convolutions. The SPEs then perform the actual edge detection using four different horizontal first order Gaussian derivative convolution kernels. After each horizontal kernel, the SPEs perform smoothing using four different zero order vertical Gaussian derivative convolution kernels. Then the SPEs take the maximum of all intermediate results and rotate the image with the reverse of the original rotation angle. Finally, the SPEs aggregate the results of all angles by again taking the maximum.
- The Edge-2D application uses a different algorithm for performing edge detection. Instead of rotating the image, it uses rotated two dimensional convolution kernels. For each of 36 different angles, the PPE generates a border around the input image. Similarly to the Edge-Rot application, four different first order Gaussian derivative convolution kernels in the rotated direction are combined with four different zero order Gaussian derivative kernels in the perpendicular direction. The SPEs perform these 16 convolutions and take the maximum of their results. Analogous to the Edge-Rot application, the SPEs aggregate the results of all angles by again taking the maximum.

The Edge-Rot and Edge-2D applications show that Cell-Space makes it easy to use different kernels on the SPE: Edge-Rot uses the geometric transformation kernel, the convolution kernel, and the maximum kernel. Cell-Space automatically balances the load among the SPEs and performs multi buffering across different kernel types. Using the same kernel across different applications is also easy: Edge-2D uses the same convolution kernel and maximum kernel as Edge-Rot. It only supplies different arguments to the convolution kernel.

Figure 7 shows the speedup of the applications on 1 to 16 SPEs, along with the speedup of the synthetic benchmark application, which runs 1024 chains of 16 jobs without persistent data. With 1 cycle/byte, the speedup of the synthetic benchmark is limited to 5 because of increasing overhead, as shown in Fig. 5. With 10 cycles/byte, linear speedup is achieved as the overhead remains constant.

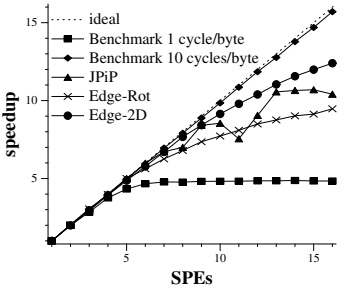


Fig. 7. Application speedup

The speedup of JPiP is limited to 11 because it suffers from load imbalance. It executes only 16 coarse-grained SPE functions in parallel, with varying compute intensities, as they decode different input files. Edge-2D and Edge-Rot achieve speedups of 9 and 12 on 16 SPEs, respectively. They are unable to achieve perfect speedup because of the overhead incurred by running fine-grained functions on the SPEs. Edge-2D performs better than Edge-Rot because its functions are more coarse-grained.

5 Related Work

Several other projects address the complexity of programming the Cell processor [12]. A challenge faced by most of these systems is that the particularities of Cell-like architectures (heterogeneous cores with local memories) make it hard to apply solutions for scheduling and communication that are based on shared memory [11,13].

Many low-level details of programming the Cell processor are handled by the Linux kernel, which has special system calls for using the SPEs. The libspe2 library provides a layer on top of the kernel and provides basic functions for using the SPEs [14]. The SPE tool chain provides a small communication library to the code that runs on the SPEs. This environment does not solve problems like load balancing, synchronization between PPE and SPE, and buffer management. These problems have to be solved by adding another layer, such as Cell-RTL.

The Charm++ Offload API [3] and the Accelerated Library Framework (ALF) [4] are similar to Cell-RTL in that they provide an API for offloading jobs to the SPEs in the Cell processor. However, Cell-RTL has many optimizations that are not present in these libraries. Moreover, Cell-RTL is more easy to use due to its simple API. An important difference with ALF is that ALF only supports data parallelism, whereas Cell-RTL supports both task and data parallelism.

Cell SuperScalar [15] and the Single Source Compiler [16] use compiler technology for using the SPEs. The user has to specify the parallel parts of the application, which are then automatically offloaded. The Single Source Compiler focuses on low-level parallelism, such as auto-SIMDization. Cell SuperScaler focuses on high-level parallelism and maintains a data flow graph of pending tasks. Although these approaches are viable for simple applications, we believe they will fail for streaming applications with complex communication patterns and corresponding data dependencies.

In the MultiCore Framework [17], the communication between the PPE and the SPEs resembles streaming. However, full streaming applications with multiple interacting components are not supported. Contrary to Cell-Space, The MultiCore Framework uses the SPEs synchronously, which results in low resource utilization.

Gedae [18] is similar to Cell-Space as it creates applications from high-level data flow specifications. It maps these specifications onto various hardware configurations, including the Cell. Although the mapping process is dynamic, Gedae statically allocates resources for the application components, whereas Cell-Space uses dynamic load balancing. Since Gedae focuses on data processing applications that do not have user interaction, we believe it does not support dynamic reconfiguration.

Several frameworks for developing streaming applications have emerged, of which the StreamIt language is the most notable example [19]. StreamIt expresses streaming applications using sequential pipeline, parallel split/join, and feed back loop primitives, which are also supported by Cell-Space. The StreamIt compiler detects parallelism in the application and maps it onto an homogeneous MPSoC [20]. This mapping is static, whereas Cell-Space uses dynamic load balancing. The StreamIt compiler supports the Cell architecture using the Multicore Streaming Layer (MSL) [5], which executes application kernels on the SPEs using static or dynamic load balancing. Contrary to Cell-Space, the main processor only runs control code and can not be used for computations in this system. Also, StreamIt does not support dynamic reconfiguration and event communication, whereas Cell-Space fully supports these advanced features.

6 Conclusions

This paper presents Cell-Space, a framework for developing streaming applications for heterogeneous multi-cores like the Cell. Developers construct applications by means of data flow components that are then scheduled on the Cell's Power core by a runtime system which offers a convenient streaming communication interface. Components may use the Cell runtime library for running jobs on the processor's synergistic processing elements. The frameworks shields developers from all low-level hardware mechanisms that are essential for performance. By carefully evaluating the relative merits of the individual mechanisms and encapsulating these under a simple API, Cell-Space greatly simplifies the development of streaming applications on the Cell without compromising performance. We have evaluated performance by means of real applications such as a 16-stream tiled display, and two edge detection algorithms.

Acknowledgments. We would like to thank Frank Seinstra for supporting the application development. This work is supported by the Dutch government's STW/Progress project 06397.

References

1. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the Cell multiprocessor. IBM Journal of Research and Development 49(4/5), 589 (2005)
2. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the Cell processor for scientific computing. In: Proc. 3rd conf. on Computing Frontiers, pp. 9–20. ACM Press, New York (2006)

3. Kunzman, D., Zheng, G., Bohm, E., Kalé, L.V.: Charm++, offload API, and the cell processor. In: Proc. Workshop on Programming Models for Ubiquitous Parallelism, Seattle, WA, USA (September 2006)
4. IBM: Accelerated Library Framework Programmer's Guide and API Reference (March 2007)
5. Zhang, X.D., Li, Q.J., Rabbah, R., Amarasinghe, S.: A lightweight streaming layer for multicore execution. In: Workshop on Design, Architecture and Simulation of Chip Multi-Processors, Chicago, IL (December 2007)
6. Nijhuis, M., Bos, H., Bal, H.E.: A component-based coordination language for efficient reconfigurable streaming applications. In: Proc. Intl. Conf. on Parallel Processing, Xi'An, China (September 2007)
7. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly, Sebastopol (2005)
8. Welsh, M., Basu, A., von Eicken, T.: Incorporating memory management into user-level network interfaces. In: Proceedings of Hot Interconnects V (August 1997)
9. Salim, J.H., Olsson, R., Kuznetsov, A.: Beyond softnet. In: Proc. 5th Annual Linux Showcase & Conference, November 2001, pp. 165–172. USENIX Association, Berkeley (2001)
10. Bos, H., de Bruijn, W., Cristea, M., Nguyen, T., Portokalidis, G.: FFPF: fairly fast packet filters. In: Proc. 6th Symposium on Operating Systems Design and Implementation (December 2004)
11. Govindan, R., Anderson, D.P.: Scheduling and ipc mechanisms for continuous media. In: SOSP, ACM SIGOPS, pp. 68–80 (1991)
12. Buttari, A., Luszczek, P., Kurzak, J., Dongarra, J., Bosilca, G.: Scop3: A rough guide to scientific computing on the playstation 3. version 0.1. Technical Report UT-CS-07-595, ICL, University of Tennessee, Knoxville (April 2007)
13. Pai, V.S., Druschel, P., Zwaenepoel, W.: Io-lite: a unified i/o buffering and caching system. ACM Transactions on Computer Systems 18(1), 37–66 (2000)
14. IBM: SPE Runtime Management Library, Version 2.2 (October 2007)
15. Bellens, P., Pérez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: Proc. 2006 ACM/IEEE Supercomputing conf., p. 86. ACM Press, New York (2006)
16. Eichenberger, A.E., O'Brien, J.K., O'Brien, K.M., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M.K., Archambault, R., Gao, Y., Koo, R.: Using advanced compiler technology to exploit the performance of the cell broadband engine™ architecture. IBM System Journal 45(1), 59–84 (2006)
17. Bouzas, B., Cooper, R., Greene, J., Pepe, M., Prelle, M.J.: Multicore framework: An API for programming heterogeneous multicore processors. In: First Workshop on Software Tools for Multi-Core Systems, Manhattan, New York, NY (March 2006)
18. Inc, G., `http://www.gedae.com`
19. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A language for streaming applications. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)
20. Gordon, M., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, CA (October 2006)

Adapting Application Mapping to Systematic Within-Die Process Variations on Chip Multiprocessors*

Yang Ding, Mahmut Kandemir, Mary Jane Irwin, and Padma Raghavan

Department of Computer Science & Engineering,
Pennsylvania State University, University Park, PA 16802, USA
{yding,kandemir,mji,raghavan}@cse.psu.edu

Abstract. Process variations, which lead to timing and power variations across identically-designed components, have been identified as one of the key future design challenges by the semiconductor industry. Using worst case latency/power assumptions is one option to address process variations. This option, while simplifying the problem, is becoming less and less attractive as its performance and power costs keep increasing. As a result, exploring options that allow the software to have knowledge about the actual latency/power consumption values is critical for future systems. Targeting systematic process variations, this paper makes two contributions. First, we discuss how we can assign threads to the cores of a chip multiprocessor (CMP) with process variations in mind and show the energy-delay product (EDP) benefits such a process variation-aware thread mapping can bring. Second, we study the benefits of varying the frequencies on a subset of the cores to increase EDP savings. We propose and evaluate integer linear programming based thread mapping schemes in both studies. While these schemes operate with profile data, they can be made to work with partial profiling as well with the help of curve fitting. We tested our schemes using both sequential and multi-threaded benchmarks from different suites and the results collected indicate that we can achieve EDP savings as much as 73.4%, with an average saving of 37.1% over a process variation agnostic scheme.

1 Introduction

Process variations have been identified as one of the key future design challenges by the semiconductor industry [3,9]. As process technology moves into the deep sub-micron regime, it is becoming increasingly difficult to control critical transistor parameters such as gate-oxide thickness, channel length, and dopant concentration. As a result, these parameters may have different values than nominal, which may, in turn, lead to both power and timing variations across identically-designed components. Such variations can occur within a chip die, called intra-die variations, and across dies, called inter-die variations. Intra-die variations are

* This research is supported in part by NSF grants 0811687, 0720645, 0720749, 0702519, 0444345 and a grant from GSRC.

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 231–247, 2009.

usually caused by channel length disparities and nondeterministic placement of dopant atoms, while inter-die variations are often due to processing temperatures and other factors [10]. As the feature size in each technology generation gets smaller, intra-die variations become increasingly important. At the same time, chip multiprocessors (CMPs) are becoming the trend in both high performance computing and embedded system domains. In the context of CMPs, intra-die variations may mean that different processor cores can have different power and performance characteristics (also called core-to-core variations).

Various parameter variations exist and continue to increase as process technologies advance. A general classification of parameter variations is provided by Unsal et al. [33]. Unlike environmental variations in voltage, temperature or input which change dynamically, most process variations are static. They usually stem from manufacturing processes including chemical mechanical polishing, lithographic exposure, resist coating, etc. Some variations are difficult to model or are unpredictable; these are referred to as random variations. Other variations which can be characterized are systematic variations. Systematic variations usually exhibit high degrees of spatial correlation. As pointed out by by Kahng et al. [21], lens aberration-induced variations become increasingly important as process margins reduce and reticle enhancement techniques improve. Our study focuses on systematic variations that are caused primarily by the lens aberrations in modern step-and-scan photolithography [13,14].

Many prior approaches have been proposed in the literature [7,16,22,26,27] to cope with process variations at the circuit and microarchitecture levels, so that their impact can be hidden from the software. For example, cores with different maximum frequencies can be uniformly clocked at the lowest frequency within a chip multiprocessor. Unfortunately, hardware approaches based on worst case assumption usually waste potential resources and their effectiveness is limited as technology scales further.

In this work, focusing on a CMP architecture, we look at the problem from a different angle by considering how to adapt the application execution to the core-to-core variations. First, given processor cores with different performance and power characteristics (due to process variations), we want to study how applications can make full use of such an architecture. Specifically, we examine the tradeoff between performance and energy consumption by focusing on the metric of energy-delay product (EDP) [17] and try to minimize its value. Our experimental results show that we can reduce the value of the EDP metric by about 11.7% on average by being careful about thread-to-core mapping when the underlying CMP exhibits systematic process variations.

We then study the potential benefits by changing the core frequencies. In this case, instead of clocking all the cores at the lowest frequency, we configure a subset of the cores to their lowest frequency, which can be higher than the globally (chip wide) lowest frequency. Different subsets of cores are considered because the number of cores needed for computations can vary as the execution progresses. Although such approach can often improve performance, its impact on the energy consumption and energy-delay product is not clear. Our experimental

results clearly indicate that a process variation-aware thread-to-core mapping (implemented using integer linear programming in this work) can save, on average, 37.1% EDP if core frequencies can be changed. While our schemes operate with profile data, they can be made to work with partial profiling as well when curve fitting (or a similar technique) is used.

The next section introduces the targeted CMP architecture and gives our major assumptions. Section 3 analyzes how to map threads to processor cores of a CMP based on several experiments that attempt to minimize the EDP under the impact of systematic within-die process variations. Sections 4 investigates design alternatives to utilize different maximum frequencies for different cores within a single-chip multiprocessor. We discusses the related work in Section 5. Finally, Section 6 summarizes our major conclusions.

2 Target CMP and Assumptions

In this study, we target at a 16-core (4×4) chip multiprocessor (CMP) architecture, as shown in Figure 1(a). Similar to the work by Humenay et al. [18], we assume the variations in transistor gate length are larger in the Y dimension than in the X dimension. This is because the critical dimension error caused by lens aberrations varies in one direction but remains constant in the other direction [21]. Consequently, the cores in different rows (also called bins) have different power and performance characteristics, whereas the cores in the same row are similar. Specifically, the cores in the top rows are faster but leakier (i.e., consume more leakage power), and the cores in the bottom rows are slower but less leaky (because the channel length is larger). We define the processor cores in one row of this CMP as a processor bin, with the bottom row in Figure 1(a) as bin 1 and the top row as bin 4.

Due to the systematic process variations in the CMP, we assume that, as compared to the frequency of cores in bin 1 (the slowest cores in the CMP), the frequencies in bin 4, bin 3 and bin 2 are 15%, 10% and 5% faster, respectively. We further assume that, as compared to the subthreshold leakage power in bin

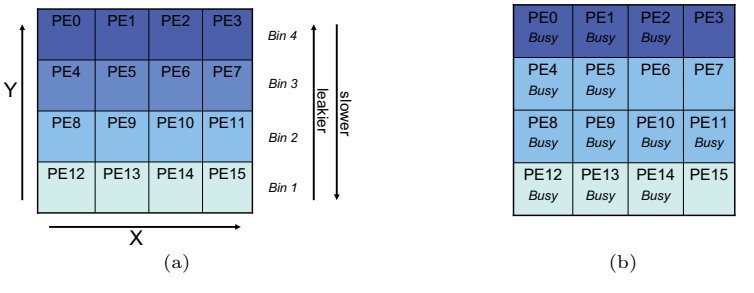


Fig. 1. (a) The targeted CMP architecture. Each row corresponds to a bin, with cores having the same power/performance characteristics. Shadings correspond to the core frequencies of the bins. (b) An example scenario of core availability when a new application is delivered to our CMP.

1, the subthreshold leakage power numbers are 60%, 40% and 20% more in bins 4, 3 and 2, respectively, because subthreshold leakage is proportional to channel length. These numbers are reasonable when compared to the results in [18], which are based on a general variation model. Note that the schemes that we propose in this paper do not depend on specific latency/power values.

Subthreshold leakage power is just one of the several components in the total power consumption for a transistor. In our work, we consider three major components including gate leakage power, subthreshold leakage power, and dynamic power. We target the $45nm$ process technology with Hi-k silicon technology and refer to [9] in estimating the proportion of each power component in the total power consumption. Gate leakage, subthreshold leakage and dynamic power are reported there to contribute approximately 10%, 20% and 70% of the total power consumption respectively in the $65nm$ process technology. The corresponding numbers are 25%, 15% and 60% in the $45nm$ process technology. However, these results did not consider the impact of using Hi-k [2] in the $45nm$ technology. To take the impact of the Hi-k technology into account, we assume that, as compared to the $65nm$ technology, the $45nm$ technology with Hi-k brings approximately 30% reduction in transistor-switching power and 10 times reduction in gate leakage power [2]. Therefore, we obtain the proportions of power consumption as 1.5% for gate leakage power, 28.5% for subthreshold leakage power, and 70% for dynamic power at a transistor level. As gate leakage power is negligible in Hi-k technology when compared to subthreshold leakage power, we consider only the subthreshold leakage power and dynamic power.

3 Mapping Application Threads to Cores

In this section, we first analyze how to map application threads to the different cores of our CMP to minimize the energy delay product, and present experimental data that illustrates the importance and potential of process variation aware application mapping. After that, we present an integer linear programming (ILP) based approach that uses profile data. Experimental results are then provided to evaluate the effectiveness of our approach.

3.1 Process Variation Aware Thread Mapping

The energy-delay product (EDP) [17] is an important metric as it captures the desire of both achieving high performance and reducing energy consumption. EDP can be calculated based on the following equations:

$$EDP = Energy \times Delay, \tag{1}$$

$$\begin{aligned} Energy &= Dynamic\ Energy + Leakage\ Energy \\ &= Dynamic\ Power \times Delay + Leakage\ Power \times Delay. \end{aligned} \tag{2}$$

Recall that we have assumed the different processor bins in our target CMP (Figure 1) have different frequencies and subthreshold leakage power. Given a

fixed number of cycles, the delay of program execution is inversely proportional to the operating core frequency. The total energy consumption consists of dynamic energy and leakage energy. Dynamic energy is proportional to CV_{dd}^2, where C is the total capacitance and V_{dd} is the supply voltage. We assume that the total capacitance is not significantly affected by the systematic process variations. Unlike prior dynamic voltage/frequency scaling approaches where V_{dd} is varied, in our targeted architecture, V_{dd} is fixed and the variation in transistor delay is caused only by process variations. In this case, the total dynamic energy is the same when executing the same program on different cores. Leakage power is proportional to $I_{leak}V_{dd}$. Here, I_{leak} is the leakage current, which mainly consists of subthreshold and gate leakage. As a result, when mapping a single thread to the CMP, the minimum EDP may be obtained by different cores depending on the ratio between dynamic energy and leakage energy. This is because the total EDP can be calculated as $(dynamic\ energy + leakage\ power \times Delay) \times Delay$. When the dynamic energy is much larger than the leakage energy, the fastest (but most leaky) core is preferred. Otherwise, the slowest (but least leaky) core should be chosen. Therefore, the assignment of threads to cores may need to be varied from thread to thread or even across the different execution phases of a given thread.

To motivate the application mapping that is aware of process variation, we implemented two sets of experiments which quantify how much benefit can be obtained if the thread-to-core assignment is aware of the underlying core-to-core variations (i.e., if the process variations are exposed to the thread mapper). Our experimental setup is explained in detail later in Section 3.3. Figure 2 shows the normalized EDP values for different single-threaded benchmarks, 10 from the

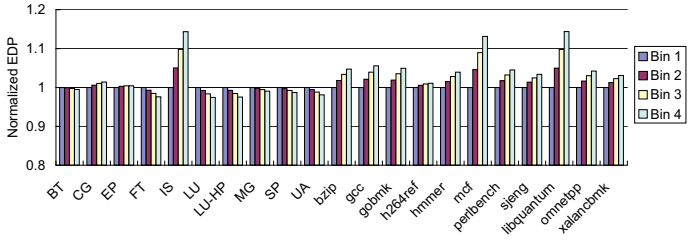


Fig. 2. Normalized EDP values when using different processor bin for single-thread applications

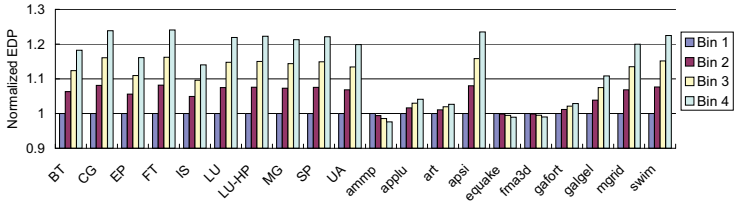


Fig. 3. Normalized EDP values when using different processor bin for applications running with four threads

NAS NPB Benchmark Suite 3.2 [4] and 11 from the SPEC 2006 CPU Benchmark Suite. Within each group of bars, the first bar represents the EDP value when the slowest but least leaky core (Bin 1) is used, and the last bar represent result when the fastest but most leaky core (Bin 4) is used. For each application, all the EDP values are normalized with respect to the result of the first bar. As we can see, for some of the benchmarks, the slowest core gives the most EDP saving (e.g., IS, mcf, libquantum); for some of them, the fastest core is favored (e.g., FT, LU, UA); and, for the rest, the choice may not make much difference. Clearly, when these applications are running on the targeted CMP, a thread-to-core mapping method that is aware of the process variations and each application's preference can bring benefit. Similarly, Figure 3 shows the normalized EDP values for different multi-threaded benchmarks (10 from the NAS NPB Benchmark Suite and 10 from the SPEC OMP 2001 Benchmark Suite), each running with four threads and thus using all four cores in a bin. All the EDP values are normalized with respect to the result shown in the first bar of each group. We see that, for most applications, the slowest cores give the most EDP saving; for the rest, the choice of cores may not make much difference. These results show that in a CMP affected by process variations, thread-to-core mapping can be important from an EDP perspective.

3.2 ILP Based Thread Mapping Scheme

We formulate the problem of the thread-to-core mapping using integer linear programming (ILP) and solve it with Xpress-MP [1], a commercial ILP solver. We assume that all threads of one application are executed at cores with the same frequency and only one thread is mapped to each core. The input of the ILP solver is the profiled information which consists of the execution time and the energy consumption values of applications. Such an ILP based thread mapping scheme can be implemented as a utility in the operating system (OS) that controls the CMP resources.

We first illustrate this approach by studying how to map the threads to cores when the number of threads is given, i.e., we make no change to the number of threads for any application execution. As we assume only one thread runs on one processor core, the number of cores assigned to each application is also fixed in this case. Supposing that there are n applications to be assigned to the four processors bins, we define a $n \times 4$ matrix s with $s(i, j)$ indicating (when the solution is found) whether the ith application has been assigned to the processor bin j. Note that our goal is to determine the value of $s(i, j)$ (once the ILP formulation is solved), which can be 0 or 1. As we also assume that all the threads of a given application are assigned to the same processor bin, we have the following two constraints:

$$\forall i, \forall j, \quad s(i,j) \in \{1, 0\}, \qquad (3)$$

$$\forall i, \quad \sum_{j=1}^{4} s(i,j) = 1. \qquad (4)$$

We use matrix D to store the execution time and matrix E to store the energy consumption for different bindings. More specifically, $D(i, j)$ and $E(i, j)$ give the

Table 1. Notations used in our ILP formulation and their meanings

Notation	Meaning
$p(i)$	number of threads for application i
n	number of application to run simultaneously
c_i	number of cores running at frequency f_i
$D(i,j)$	execution time of application i when running on cores at frequency f_j
$E(i,j)$	energy consumption of application i when running on cores at frequency f_j
$s(i,j)$	whether application i is running on cores at frequency f_j
$Bound_i$	performance bound for application i

performance delay and total energy consumption, respectively, when the threads of the ith application are assigned to run in bin j of our CMP. As mentioned above, these values are collected through profiling. When complete profile data is not available, we can also use partial profiling and apply curve fitting methods to estimate the unknown values.

Once we have the matrices D and E set up, an optimization objective can be easily defined. For example, if the objective is to minimize the summation of the EDP values of all the applications, we can program the ILP solver to minimize the following objective function:

$$\sum_{i=1}^{n}\sum_{j=1}^{4} D(i,j) \times E(i,j) \times s(i,j). \tag{5}$$

Recall that $D(i,j)$ and $E(i,j)$ are constants and obtained from profiling. In the above discussion, we assumed that the number of threads for each application is given. This number can be obtained separately by choosing the optimal number of threads with respect to the optimization objective. If desired, we can also integrate this search dimension into our ILP formulation of the problem (i.e., our formulation can be modified to give the ideal number of cores as well). If we use $p(i)$ to represent the number of threads (i.e., the number of processor cores under our assumption) used to execute the ith application, the following inequalities need to be added to our resource constraints:

$$\sum_{i=1}^{n} p(i) \leq 16, \tag{6}$$

$$\forall j, \quad \sum_{i=1}^{n} p(i) \times s(i,j) \leq 4. \tag{7}$$

While the first constraint limits the total number of cores in the CMP, the second one indicates that there are four cores in each processor bin. In this case, the matrices D and E need to be extended to capture the execution time and energy consumption for an application at different thread counts. A summary of the notations used in our ILP formulation is given in Table 1. Note also that it is not difficult to extend our ILP formulation to larger CMPs with more bins.

3.3 Experimental Setup

We simulate the CMP architecture given earlier in Figure 1 using the Simics toolset [5]. Simics is a multiprocessor simulator that can be used to perform full

Table 2. Our major simulation parameters and their default values

Parameter	Value
Number of Cores	16
Maximum Frequency in Bin 1 (f_1)	2GHz
Maximum Frequency in Bin 2 (f_2)	2.1GHz
Maximum Frequency in Bin 3 (f_3)	2.2GHz
Maximum Frequency in Bin 4 (f_4)	2.3GHz
L1 Data Cache	64K, 2-way, 2 banks
L1 Instruction Cache	64K, 2-way, 2 banks
Unified L2 Cache	4MB, 16-way, 2 banks
Process Technology	45nm

system simulation. The operating system running on each core in the simulator is Solaris 9. Table 2 gives the major simulation parameters and their default values. Each processor core has its own private instruction and data L1 caches, and all cores share a unified on-chip L2 cache. We implemented the MESI protocol to provide cache coherency. To calculate energy consumption, we use Wattch-like [12] models for the processor cores; cache energy consumptions are calculated using CACTI 5.0 [32].

In our experiments, we use the NAS Parallel Benchmark Suite 3.2 [4] and the SPEC Benchmark Suites. The NAS Suite provides several implementations, including serial, MPI and OpenMP. These codes represent computations that are used in a large variety of modeling and simulation applications based on finite element, finite difference and spectral methods. We use the serial implementation for the single-threaded applications. For the multi-threaded applications, we use the OpenMP based implementations [20]. To make simulation time affordable, we use the class W inputs and run all benchmarks to completion. For the SPEC benchmarks, we use SPEC CPU 2006 for single-threaded applications and SPEC OMP 2001 for multi-threaded applications. Simulating a SPEC benchmark to completion with reference data sets requires too much time in our simulator. Therefore, for the single-threaded benchmarks, we simulate 1 billion instructions after the warm-up of 5 billion instructions; for the multi-threaded benchmarks, we simulate several major iterations after warm-up to make sure the same amount of workload is simulated on different parallel executions.

3.4 Experimental Evaluation

We consider four groups of multi-threaded applications and distribute the 16 cores in our CMP across these applications. Each group has four applications and each application has four threads. If the mapping method has no knowledge about application characteristics or core-to-core variations, each application can be assigned to any arbitrary bin. Figure 4 shows the EDP results under different assignments. Points on the x-axis show the different workload mixes. Values in each group are normalized to the fourth bar. The results obtained using the ILP based scheme explained earlier correspond to the first bar in each group. The second, third, and the fourth bars represent, respectively, the best, average and worse cases in terms of EDP of all the possible assignment. We can see that the EDP savings achieved by our ILP based approach are essentially the same as

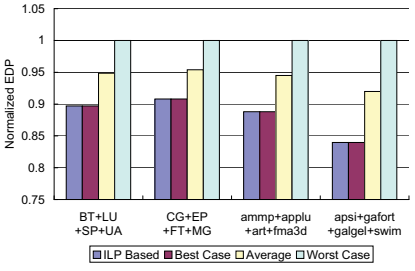


Fig. 4. Normalized EDP values when using different mapping methods. Each group of bars is for a set of 4 applications running simultaneously on the same CMP.

those of the best case. We also observe that the performance and EDP overhead incurred by the ILP solver is negligible. Our experiments with partial profile data also successfully identified the trends through a linear curve fitting method and generated the optimal results. In fact, as long as we have the profile data for each application in two different bins, the optimal results can be obtained based on fitted results for executions in other bins. This is because, for the benchmarks we tested and shown in Figure 3, the EDP change is monotonic when scheduled on cores with decreasing frequencies.

4 Changing Bin Frequencies

In practice, cores with different maximum frequencies are usually uniformly clocked at the lowest frequency within a CMP. This implies setting all the cores in our targeted CMP to the frequency of bin 1 (the one with the lowest frequency). An alternative would be to allow different bins to have different frequencies and to downgrade bins that can run at higher frequencies to a lower frequency if doing so is beneficial; still, all cores in the same bin run at the same frequency. We assume the maximum frequency for bin i is f_i. In this case, cores in bin 1 can only run at frequency f_1; cores in bin 2 can run at f_2 or f_1 (where f_2 is 5% faster than f_1), etc.

Figure 5 shows the performance, energy, and EDP results for the *swim* benchmark in SPEC OMP 2001 when using a different number of cores. To keep the search space reasonably small, we assume that all the cores assigned to the same application operate at the same frequency and no more than one thread runs on one processor core. As a result, when an application uses four threads, possible frequency values can be f_1, f_2, f_3, or f_4. However, when 16 threads are used, the frequency can only be f_1 after downgrading all the cores in different bins to the lowest frequency. Informally, this means we can use a large number of slower cores, or a small number of faster cores. From Figure 5(a), we see that the performance generally improves as the number of threads used increases, though it may saturate beyond a certain point. As expected, the performance is also better when faster cores are used. When considering the energy consumption, we see from Figure 5(b) that the energy consumption slightly increases when

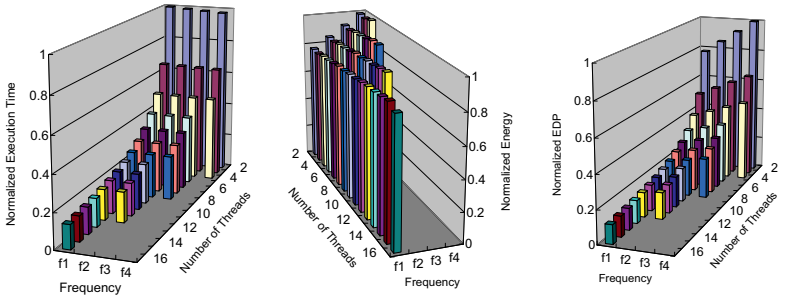


Fig. 5. Performance, energy, and EDP results (from left to right) for the *swim* benchmark when using different number of cores, assuming reconfigurable processor bins

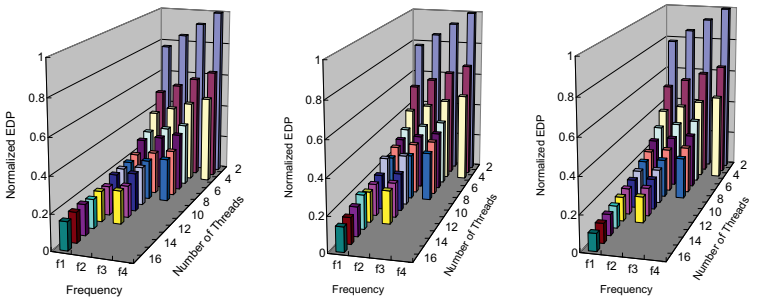


Fig. 6. EDP results for the *FT*, *MG* and *wupwise* benchmarks (from left to right) when using different number of cores, assuming reconfigurable processor bins

the number of threads increases. This is because there are more components consuming leakage power when more threads and more cores are used.[1] When the number of threads is fixed, using faster but leakier cores consumes more energy. The reason is that the frequency variation is far less than the leakage power variation in our targeted CMP. We also observe that the difference when changing the number of threads is not as significant as that with changing core frequencies. As a result, a clear trend can be observed in the EDP results in Figure 5(c). With some exceptions, the EDP values generally increase as the operating frequency increases and decrease as the number of threads increases.

4.1 ILP Based Thread Mapping Scheme

We can see clear trends from the results for the *swim* benchmark in Figure 5 and several other benchmarks shown in Figure 6. An important question to answer now is how to build such graphs for different applications at runtime. Note that

[1] We assume that when a core is idle it is shut off (V_{dd} is disconnected) and thus not consuming leakage power.

we can first obtain some of the results based on the profiled data such as the statistics from the previous executions. After that, we can apply curve fitting methods to predict the unknown results for different settings. Once we have such a mechanism in place, different optimization problems can be solved using an ILP solver, similar to the approach discussed in Section 3. For example, besides simply minimizing the energy-delay product, we can try to minimize the energy consumption under certain performance bound by using the same set of variables as before and adding the following constraint:

$$\forall i, \sum_{j=1}^{4} D(i,j) \times s(i,j) \leq Bound_i. \tag{8}$$

Here, $Bound_i$ is the execution latency bound for the ith application. Thus, the performance requirement is satisfied when the application uses cores from the chosen processor bin. Assuming that we want to minimize energy consumption, we can express our objective function (to minimize) as follows:

$$\sum_{i=1}^{n} \sum_{j=1}^{4} E(i,j) \times s(i,j). \tag{9}$$

Compared to the ILP formulation in Section 3, the number of cores running at each frequency is no longer fixed at 4. We assume the number of cores at frequency f_i is c_i. Thus, if the core frequencies (and voltages) are not changed, we have $c_1 = c_2 = c_3 = c_4 = 4$. In the case where we can change the frequency of each processor bin as explained above, the possible values for c_i are different but they all should be multiple of fours. For example, c_1 can be 4, 8, 12 or 16 because different processor bins can be downgraded to frequency f_1; but c_4 can only be 4 or 0 because only bin 4 can run at frequency f_4. In summary, the following constraints should be satisfied for c_i:

$$\sum_{i=1}^{4} c_i = 16, \tag{10}$$

$$\forall j, \quad \sum_{i=1}^{n} p(i) \times s(i,j) \leq c_j, \tag{11}$$

$$c_1 \in \{4, 8, 12, 16\}, \tag{12}$$

$$c_2 \in \{0, 4, 8, 12\}, \tag{13}$$

$$c_3 \in \{0, 4, 8\}, \tag{14}$$

$$c_4 \in \{0, 4\}. \tag{15}$$

The first two of these constraints limit the total number of cores and the number of cores at each frequency. The rest just specify the possible number of cores that can be used at each frequency level.

The thread-to-core mapping method needs to dynamically decide the mapping of processor cores to different applications at runtime. When a new application is delivered to our targeted CMP for execution, there are two choices for core allocation. The first choice is to use only the available cores without affecting (disturbing) any other applications, i.e., it is non-preemptive. The second choice is to obtain an optimized global allocation by re-assigning cores to each application, which we call preemptive mapping. Note that preemptive mapping can also require a change in the frequencies of all cores of a given bin.

We now discuss an example to illustrate the difference between the non-preemptive mapping and preemptive mapping. As illustrated in Figure 1(b), we assume the four processor bins are currently running at frequencies $f4$, $f2$, $f2$ and $f1$ (i.e., $c_4 = 4$, $c_3 = 0$, $c_2 = 8$, and $c_4 = 4$). We further assume that the number of unoccupied cores in the four bins are 1, 0, 2 and 1, starting from bin 1. In other words, the new application to be executed can choose resource from one core running at frequency $f4$, two cores running at frequency $f2$, and one core running at frequency $f1$. Let us first consider the non-preemptive mapping scheme. If the application is single-threaded, we can choose a core at one of the three frequencies. If the application uses two-threads, we can only choose the two cores running at frequency $f2$ (recall that we assume no more than one thread is allowed to run on each core and the threads of one application must run on same type of cores). In contrast, if preemptive mapping is allowed, we can reclaim some processor cores from other running applications and assign them to the new application. Both non-preemptive and preemptive mappings are handled using ILP. The non-preemptive mapping is much easier to implement and less costly than the preemptive mapping, but it often settles on a globally sub-optimal solution. Also, the mapping action may be delayed in the preemptive mapping, since the processor cores may not be reclaimable at arbitrary point of the program executions. In this work, we do not discuss the mechanism to reclaim cores and only apply our preemptive mapping scheme on application entry and exit points to investigate its potential benefits.

4.2 Experimental Evaluation

We first studied the behavior of the non-preemptive thread mapping by comparing our ILP based approach with three alternate schemes. The first alternative is to use the processor bin with the most unoccupied cores and break possible ties through random selection (this option is referred as MOST). The other two alternatives are to use the cores with the lowest or highest frequency (which are called SLOWEST and FASTEST). Figure 1(b) shows a possible scenario of the available cores (i.e., the cores that are not marked as "busy") when an application is about to be mapped. There are many such scenarios, so we investigated all the possible scenarios and compared the four different mapping methods (our ILP based mapping, MOST, SLOWEST, and FASTEST). The results are plotted in Figures 7 and 8 for the *FT*, *MG*, *swim* and *wupwise* benchmarks.

Figure 7 represents the probability that a thread mapping scheme generates the best EDP results out of all the possible scenarios (each of which is similar to the one shown in Figure 1(b)). We see that our ILP based thread mapping scheme can always make the best decision and is therefore better than the other mapping schemes tested. Figure 8 illustrates the average and maximum (across all the scenarios) of the normalized EDP values under the different thread mapping schemes for each benchmark. In our tested benchmarks, the mapping methods FASTEST and SLOWEST generate much worse results than the ILP based approach. The results are reasonably good on average when using the processor bin with the most unoccupied cores (referred to as MOST in the figures), though

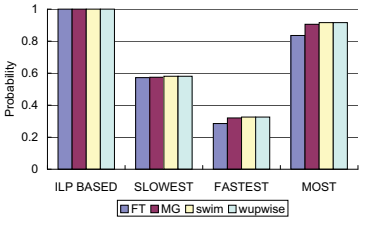


Fig. 7. Probability to generate the best EDP results under the different thread mapping schemes

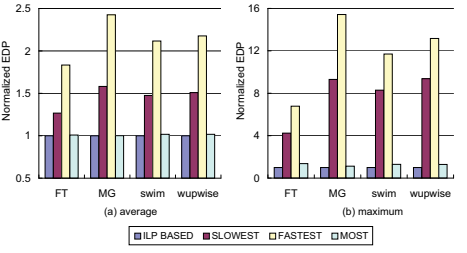


Fig. 8. Average and maximum in the normalized EDP results under the different thread mapping schemes. For each benchmark, all the bars are normalized with respect to the first bar.

its results can be as much as 36% worse than the ILP based scheme in some cases. The main reason for this is that the parallel applications usually scale very well when the number of cores is relatively small and using all the available cores reduces the EDP as well.

We now consider a group of four applications running on our target CMP architecture and explore the benefits of the preemptive thread mapping scheme over the non-preemptive scheme. We assume that three applications were occupying all the cores in the optimal way and one has finished execution before the fourth application is delivered to the CMP. With non-preemptive mapping, the fourth application can only use the cores left idle by the third application. By comparison, in preemptive mapping, all the cores can be reassigned and frequencies can be re-tuned at the bin granularity. In this process, different thread migration approaches [28,34] can be used to move threads from one core to another and the machine state may need to be saved using checkpointing [11,30]. The actual migration and checkpoint strategies that can be employed are orthogonal to the main focus of this paper. Therefore, we do not discuss them further here.

Figure 9 shows the percentage EDP savings when using preemptive thread mapping instead of non-preemptive mapping. The four benchmarks used in this experiment are FT, MG, *swim*, and *wupwise*. There are 12 scenarios to consider because each application can be the new-comer or the finishing one. For example, one scenario would be that FT and MG are running, and *swim* has finished just before *wupwise* arrives. We see from Figure 9 that, on average, the preemptive mapping scheme can bring 37.1% extra EDP savings. Note also that, in four cases, the preemptive mapping scheme generates the same result as the non-preemptive one, and thus, no bar is plotted for these cases.

So far, we have assumed the complete performance and energy profile statistics are available when making the thread-to-core mappings. We now vary the amount of profile information available to us before mapping threads to cores, and apply a linear curve fitting method based on the partial profile data to help making the decisions. Figure 10 shows the EDP results normalized with respect to the results achieved using our non-preemptive thread mapping scheme, when

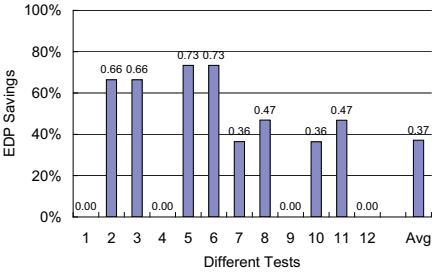


Fig. 9. Percentage of the EDP savings for the preemptive mapping over the non-preemptive mapping scheme

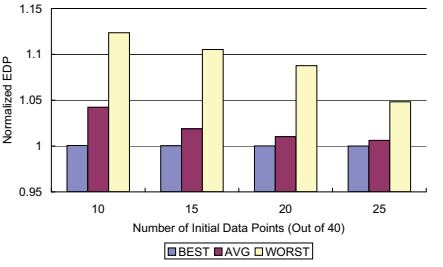


Fig. 10. Normalized EDP results with partial profiling and curve fitting for our ILP based method

we randomly choose the initial data points in the search space and then apply a linear curve fitting method. The x-axis indicates the total number of initial data points used. Each group of bars gives the minimum, average and maximum EDP results that can be obtained based on 1000 different sets of initial data points. We can see from this plot that the results are still promising with partial profiling data and improve as more profiling data becomes available. Note that the results in the best case are all within 1% of the optimal. Their absolute values also decrease as the number of initial data points increases. Overall, these results indicate that our approach works very well even with partial profile data.

5 Related Work

Most of the previous efforts try to eliminate or mitigate the impact of systematic intra-die process variations [10,18,33]. In contrast, this paper studies how to adapt application execution to such variations in order to optimize the performance, energy consumption, or the tradeoff between them. Donald and Martonosi [15] proposed an analytical method to meet the power-performance requirements under process variations. Their approach estimates tolerable process variations and adaptively decides the processor cores to be turned off. The major difference between their work and ours is that we focus on thread-to-core mapping and target a CMP with specific systematic process variations.

Humenay et al. [18] focused on the same source of variations as our study (i.e., non-uniformity in the lithographic exposure field) and focused on the core-to-core variations due to spatially correlated intra-die process variations. They investigated the tradeoffs between static performance asymmetry due to frequency variation and dynamic performance asymmetry due to thermal throttling. They concluded that both hardware and software techniques are necessary to address the problems caused by core-to-core variations. Roberts et al. [29] investigated a scheduling scheme that is aware of the transistor wear-out and presented theoretical results for a set of streaming applications. Teodorescu and Torrellas [31] also focused on within-die process variation and proposed a variation-aware algorithm for both thread scheduling and dynamic voltage/frequency scaling. Their

work only considered single-threaded applications, whereas we specifically focus on parallel applications and consider changing the number of threads during the course of execution.

One of the obvious results of core-to-core variations is performance asymmetry. Balakrishnan et al. [6] studied the impact of performance asymmetry in a multicore architecture. They focused on the predictability and scalability of performance when commercial workloads are executed on a multicore system where individual cores have different performance characteristics. Their results suggest that the application needs to be aware of performance asymmetry and some degree of performance asymmetry is beneficial.

Employing cores with different characteristics is not new in designing CMP architectures. Heterogeneous CMPs have been studied in the literature. Kumar et al. [25] demonstrated that, as compared to a conventional homogeneous CMP, significant performance improvement can be obtained by matching the different jobs of a diverse workload with the different cores designed with various complexities. In another work [24], Kumar et al. also showed how to customize each core to a different subset of application characteristics. Becchi and Crowley [8] proposed dynamic thread assignment and exploited thread migration between cores of a heterogeneous multiprocessor. The heterogeneous cores are usually designed to be different in order to satisfy the various requirements of specific applications. In contrast, the core asymmetry in our study is caused by process variations. Also, most of the previous work focused on performance, with only a few addressing power issues, whereas we choose EDP to optimize the tradeoff between performance and power/energy consumption.

While the parameters in conventional hardware design are usually fixed, software can behave very differently and have various demands. To accommodate such software diversity, several recent studies propose reconfigurable chip multiprocessor architectures. Ipek et al. [19] presented a complete hardware solution which supports multiple simple cores to dynamically fuse into larger and more powerful processors. Kim et al. [23] evaluated a different architectural approach, called composable lightweight processors, which achieve full composability based on a non-standard ISA and also allow aggregating simple cores together. The major difference between our work and these efforts is that we focus on adapting the thread mapping to process variations and only study reconfiguration of the core frequencies and voltages.

6 Conclusions

In this paper, we study how to adapt application execution to spatially-correlated systematic process variations, where different cores of a CMP can have different performance and power characteristics. We first explore the thread mapping schemes to optimize performance, energy consumption and energy-delay product. We then study the benefits by reconfiguring a subset of the cores to their lowest frequency based on specific application behaviors, instead of clocking all the cores at the globally lowest frequency. We propose integer linear programming

based mapping schemes in both studies and the experimental results show that the thread mapping method that are aware of the core-to-core variation can successfully optimize performance, energy consumption, and energy-delay product globally. We also observe that significant reduction in energy-delay product can be obtained using different reconfigurations of the core frequencies.

References

1. Applications of optimization with Xpress-MP, http://www.dashoptimization.com/
2. Intel 45nm hi-k silicon technology, http://www.intel.com/technology/architecture-silicon/45nm-core2/
3. International technology roadmap for semiconductors (2005), http://public.itrs.net/
4. NAS NPB benchmarks 3.2, http://www.nas.nasa.gov/Software/NPB/
5. Virtutech Simics 3.0, http://www.virtutech.com/
6. Balakrishnan, S., Rajwar, R., Upton, M., Lai, K.: The impact of performance asymmetry in emerging multicore architectures. In: Proceedings of International Symposium of Computer Architecture (2005)
7. Banerjee, N., Karakonstantis, G., Roy, K.: Process variation tolerant low power DCT architecture. In: Proceedings of Design, Automation and Test in Europe (2007)
8. Becchi, M., Crowley, P.: Dynamic thread assignment on heterogeneous multiprocessor architectures. In: Proceedings of Computing Frontiers (2006)
9. Borkar, S.: VLSI Design Challenges for Gigascale Integration. In: Proceedings of International Conference on VLSI Design (2005)
10. Borkar, S.: Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. IEEE Micro. 25(6) (2005)
11. Bronevetsky, G., Marques, D., Pingali, K., Szwed, P., Schulz, M.: Application-level checkpointing for shared memory programs. In: Proceedings of Architecture Support for Programming Languages and Operating Systems (2004)
12. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: a framework for architectural-level power analysis and optimizations. In: Proceedings of International Symposium of Computer Architecture (2000)
13. Cain, J.P., Zhang, H., Spanos, C.J.: Optimum sampling for characterization of systematic variation in photolithography. In: Proceedings of SPIE, vol. 4689 (2002)
14. Choi, M.: Modeling of deterministic within-die variation in timing analysis, leakage current analysis, and delay fault diagnosis. School of Electrical and Computer Engineering Theses and Dissertations, Georgia Institute of Technology (2007)
15. Donald, J., Martonosi, M.: Power efficiency for variation-tolerant multicore processors. In: Proceedings of International Symposium of Computer Architecture (2006)
16. Fetzer, E.S.: Using adaptive circuits to mitigate process variations in a microprocessor design. IEEE Design & Test 23(6) (2006)
17. Gonzalez, R., Horowitz, M.: Energy dissipation in general purpose microprocessors. In: Proceedings of Internationl Symposium on Low Power Electronics (1995)
18. Humenay, E., Tarjan, D., Skadron, K.: Impact of process variations on multicore performance symmetry. In: Proceedings of Design, Automation and Test in Europe (April 2007)

19. Ipek, E., Kirman, M., Kirman, N., Martinez, J.F.: Core fusion: accommodating software diversity in chip multiprocessors. In: Proceedings of International Symposium of Computer Architecture (2007)
20. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011 (1999)
21. Kahng, A.B., Park, C.-H., Sharma, P., Wang, Q.: Lens aberration aware timing-driven placement. In: Proceedings of Design, Automation and Test in Europe (2006)
22. Kang, K., Kim, K., Roy, K.: Variation resilient low-power circuit design methodology using on-chip phase locked loop. In: Proceedings of Design Automation Conference (2007)
23. Kim, C., Sethumadhavan, S., Govindan, M., Ranganathan, N., Gulati, D., Burger, D., Keckler, S.W.: Composable lightweight processors. In: Proceedings of International Symposium on Microarchitecture (2007)
24. Kumar, R., Tullsen, D.M., Jouppi, N.P.: Core architecture optimization for heterogeneous chip multiprocessors. In: Proceedings of Parallel Architecture and Compilation Techniques (2006)
25. Kumar, R., Tullsen, D.M., Ranganathan, P., Jouppi, N.P., Farkas, K.I.: Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In: Proceedings of International Symposium of Computer Architecture (2004)
26. Liang, X., Brooks, D.: Mitigating the impact of process variations on processor register files and execution units. In: Proceedings of International Symposium on Microarchitecture (2006)
27. Meng, K., Joseph, R.: Process variation aware cache leakage management. In: Proceedings of International Symposium on Low Power Electronics and Design (2006)
28. Milojičić, D.S., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. ACM Computer Surveys 32(3) (2000)
29. Roberts, D., Dreslinski, R., Karl, E., Mudge, T., Sylvester, D., Blaauw, D.: When homogeneous becomes heterogeneous: Wearout aware task scheduling for streaming applications. In: Workshop on Operationg System Support for Heterogeneous Multicore Architectures (2007)
30. Shafi, H., Speight, E., Bennett, J.K.: Raptor: integrating checkpoints and thread migration for cluster management. In: Proceedings of International Symposium on Reliable Distributed Systems (2003)
31. Teodorescu, R., Torrellas, J.: Variation-aware application scheduling and power management for chip multiprocessors. In: Proceedings of International Symposium of Computer Architecture (2008)
32. Thoziyoor, S., Muralimanohar, N., Jouppi, N.P.: CACTI 5.0. Hewlett-Packard technical report HPL-2007-167 (2007), `http://quid.hpl.hp.com:9082/cacti/`
33. Unsal, O.S., Tschanz, J.W., Bowman, K., De, V., Vera, X., Gonzalez, A., Ergin, O.: Impact of parameter variations on circuits and microarchitecture. IEEE Micro. 26(6) (2006)
34. Veldema, R., Philippsen, M.: Near overhead-free heterogeneous thread-migration. In: Proceedings of International Conference on Cluster Computing (2005)

Accomodating Diversity in CMPs with Heterogeneous Frequencies

Major Bhadauria[1], Vince Weaver[1], and Sally A. McKee[2]

[1] Computer Systems Lab
School of Electrical and Computer Engineering
Cornell University
{major,vince}@csl.cornell.edu
[2] Department of Computer Science and Engineering
Chalmers University of Technology
sallyamckee@gmail.com

Abstract. Shrinking process technologies and growing chip sizes have profound effects on process variation. This leads to Chip Multiprocessors (CMPs) where not all cores operate at maximum frequency. Instead of simply disabling the slower cores or using guard banding (running all at the frequency of the slowest logic block), we investigate keeping them active, and examine performance and power efficiency of using frequency-heterogeneous CMPs on multithreaded workloads. With uniform workload partitioning, one might intuitively expect slower cores to degrade performance. However, with non-uniform workload partitioning, we find that using both low and high frequency cores improves performance and reduces energy consumption over just running faster cores. Thread scheduling and workload partitioning naturally play significant roles in these improvements. We find that using under-performing cores improves performance by 16% on average and saves CPU energy by up to 16% across the NAS and SPEC-OMP benchmarks on a quad-core AMD platform. Workload balancing via dynamic partitioning yields results within 5% of the overall ideal value. Finally, we show feasible methods to determine at run time whether using a heterogeneous configuration is beneficial. We validate our work through evaluation on a real CMP.

1 Introduction

With single core chips, CPU manufacturers detect under-performing processors at production time and often sell them as less-powerful, value-priced CPUs. In the chip multiprocessor (CMP) domain, one current trend is to disable under-performing cores. As the number of on-chip cores grows to 16 or more, process variation makes it more difficult to achieve uniform, high-frequency operation. Bowman and Meindl [6] find that process variation on chip can lead to frequency reductions approximately equivalent to one step "backwards" in process technology. Heterogeneous frequencies can be designed to accommodate a specific power envelope, with high frequency cores for serial program portions, and low frequency cores for parallel portions. We examine benefits of running under-performing cores at lower frequencies, instead of disabling them.

Intuitively, with uniform workload partitioning, one might expect low-frequency cores to reduce the performance of high-frequency cores. Additionally, one might expect computing efficiency to be better when solely running faster cores. In contrast,

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 248–262, 2009.

we find that scheduling workloads non-uniformly can improve both performance and energy consumption. Even if yields could achieve uniformly high frequencies for all cores on a CMP, we still make a case for heterogeneous processing, even if only for the embedded market (where power is a first order design concern). Our chief concern here is performance, but the energy improvements of using heterogeneous cores wisely are not insignificant. This latter observation will grow in importance with increased numbers of cores and limited on-chip power budgets. Our findings may already be known to industrial practitioners, but we have yet to see such results made public.

We measure execution time to quantify improvement in delay on high-performance, multithreaded scientific codes. We allocate differing workloads by executing more iterations of parallelized loops on faster cores. We answer four sets of questions:

1. What are the potential speedups from utilizing significantly slower processors?
2. What are the impacts and overheads of dynamic scheduling techniques? How dependent are performance gains on the scheduling technique? How equally are instructions divided between cores?
3. What are the power/performance tradeoffs of heterogeneous versus homogeneous chips? Are fewer threads more efficient than adding extra threads on slower cores? Can energy be saved by using heterogeneous cores?
4. How do we dynamically determine if using slower cores improves performance?

We execute multithreaded benchmarks from NAS and SPEC OMP on real hardware, measuring execution time and total power consumption. Our results show that using diverse cores can deliver significant speedups and reduce total energy consumption for most applications. The added overhead of slower cores is offset by the energy costs of shared resources such as caches, buses, and main memory. This will be important in future CMP systems, since process variations limit the feasibility of uniform cores for larger-scale CMPs.

2 Background

Here we discuss static and dynamic scheduling for loop-oriented, multithreaded codes.

2.1 Static Scheduling

OpenMP parallelizes loop bodies of Fortran or C programs marked by PARALLEL pragmas. Code in parallel regions contains independent iterations that can be computed by separate threads. Each processor is usually assigned one thread, although multiple threads may be given to cores supporting simultaneous multi-threading [15] or multiprogramming. By default, the OpenMP scheduler assumes loop iterations require equal amounts of computation, and thus it statically assigns an equal number of iterations to each thread. We refer to these as chunks (Eq. (1)) (akin to blocks of iterations for scheduling parallel loops on symmetric multiprocessors [SMPs]). If some loops run longer than others during a workload chunk, a load imbalance occurs, and one or more threads finishing before others. Since ends of parallel sections contain implicit barriers, faster threads wait at barriers, idling until the slowest thread finishes.

$$static\,workload\,chunk = \frac{total\,iterations}{number\,of\,threads} \tag{1}$$

```
#pragma omp parallel for
BEGIN PARALLEL FOR
    A[i]=B[i]*B[i];
END PARALLEL FOR
```

Fig. 1. Example Parallelized For Loop

The first chunk is assigned to the first thread, the second chunk to the next, and so forth, until all chunks have been assigned. Each thread can amortize costs of fetching data into cache via multiple accesses to those data used in subsequent loop operations and iterations. The overhead cost of partitioning the workload only occurs once.

Figure 1 illustrates a simple example. The code reads array `B`, computes a result, and writes that to array `A`. When a value at index `i` is fetched, other elements in the same cache line are also fetched. If the arrays are dense for a block-cyclic partitioning of iterations, memory costs of fetching elements `i+1`, `i+2`, ..., can be amortized over multiple iterations. Alternatively, for cyclic partitioning, in which iterations are assigned round-robin to threads, false sharing occurs if threads work on adjacent indices. This is the same problem as loop scheduling for SMPs, but in a different context (we're advocating block-cyclic over cyclic assignments of iterations to cores).

2.2 Dynamic Scheduling

For static scheduling, chunk sizes and number of chunks are pre-chosen based on number of threads. For dynamic scheduling, the user can specify chunk size and the number of chunks can exceed the number of threads. Chunks are assigned round-robin to available threads. New chunks are allocated when previous chunks complete. This tries to overcome the main drawback of static scheduling — the potential for load imbalance — by using smaller chunk size that are dynamically assigned to cores.

Guided scheduling is a variation of dynamic scheduling, where chunk sizes are not fixed, and the user can specify a lower-bound for sizes. Chunk sizes can be larger but never smaller than the specified lower-bound. The run-time library starts by allocating larger chunk sizes, and keeps decreasing sizes until it reaches the lower limit specified by the user (or a chunk size of one iteration if no lower-bound is given).

The dynamic approach minimizes false sharing and amortizes memory costs for accessing data touched multiple times within and across iterations. This approach assumes many small loop iterations and chunk sizes sufficiently large to make scheduling overhead negligible, but remains sufficiently flexible to balance work between overloaded and idle cores. We examine this strategy for loop-oriented benchmarks, noting what problems arise and identifying counter measures. We engineer our implementation for loops with different computation overheads and underlying thread hardware operating at different speeds. Latency to memory remains unchanged, but computation overhead per loop increases for slower cores. All changes are compiler-agnostic with respect to the underlying hardware, such that the workload can be appropriately allocated at run time, rather than at compile time.

Programmers typically parallelize the outer-most loop with nested loops. When programs contain many nested loops, load-balancing for just the outer-most loop may be insufficient. For example, *applu* and *lu* both contain many nested loops. Parallelizing

the outer-most loop may not be a sufficiently small granularity for balancing workloads between heterogeneous cores. When migrating to heterogeneous systems, hand optimizations (outside the scope of this paper) can help balance the system.

3 Related Work

Practitioners compensate for variations by correcting them or by finding ways (in hardware or software) to tolerate them. Process variations cause CPUs to reach different maximum frequencies for given operating voltages, and even though CMPs are more tolerant of variations [5], the problem remains important. Variations can be corrected in multiple ways [20,18,19], but most common is increasing operating voltage for underperforming cores, which increases power consumption. Humenay et al. [11] find accommodating variation via voltage increases can consume 166% more power than regular cores and requires separate voltage islands per core. Donald et al. [7] study techniques to turn cores on and off, adding scheduling complexity. Unlike our study, other work examines cores running at one frequency but using differing amounts of power.

Variation accommodation falls under two domains: effectively utilizing heterogeneous systems, and reducing power consumption through DVFS (Dynamic Voltage and Frequency Scaling) within CMPs or computing clusters without degrading performance. Under the first, researchers use multithreaded programs for CMPs of different processors on chip. Liu et al. [16] achieve scalable speedups with different processors working in unison by extending OpenMP and hand optimizing codes.

Wong et al. [21] analyze load balancing of a heterogeneous symmetric multiprocessor (SMP) server and a cluster of SGI and Intel machines. These run parallel applications on different architectures simultaneously. They examine static scheduling via OpenMP threads and dynamic scheduling using iteration profiling. They find that a mixture of techniques improves performance but increases profiling and complexity. Balakrishnan et al. examine multithreaded program performance on a prototype SMP running processors at different frequencies [4]. They find that exposing asymmetry to the OS and programmer enables performance improvements. This SMP scalability and predictability study assumes separate chips, and thus the study does not address power implications and memory effects of CMP systems. This system has no shared caches, thus chip-to-chip latencies affect thread scaling and processor communication assumptions. Unfortunately, individual processor workloads are not revealed, making it hard to discern sources of performance bottlenecks. Both studies fail to explore power efficiency of such systems, which we specifically target.

Kadayif et al. [14] leverage heterogeneity between threads to reduce power consumption. They perform DVFS on cores independently, slowing faster threads with minimal performance degradation. Isci et al. [12] find improvements in computing efficiency to be worth the added overhead of per-core voltage islands, but the study uses a multiprogrammed workload of single threaded benchmarks in unison on a CMP. In contrast, Herbert et al. [10] use multi-threaded benchmarks and find that per-core DVFS is not worth the complexity of the independent voltage islands compared to chip wide DVFS. The preferred approach appears to depend on the desired workloads. Since ours consist of multi-threaded benchmarks, we target the infrastructure without per core voltage

islands. Across clusters of processors, Ge et al. [9] make it clear that DVFS is worth the implementation cost, as they achieve power reductions without significant performance loss. The common thread among these studies has been the use of DVFS to achieve energy savings while the processor is memory bound [13] to avoid significant performance degradations.

We believe ours to be the first work using real hardware to examine performance and energy for frequency heterogeneous CMPs on multithreaded, shared memory codes.

4 Experimental Setup

We compile with `-openmp` and `-static` and Intel C and Fortran compilers. For NAS v3.2 [3], we use class B large inputs outlined in Table 1. These fit in main memory, but are large enough to provide significant work per thread. We omit *IS*: it fails to run for larger input sets. For the SPEC OMP benchmarks [17], we use training inputs, since they provide reasonable workloads and complete in reasonable time. Native execution times varies from about one-ten minutes, depending on benchmark. We use Linux 2.6.24.2 and the perfmon2 library [8] to gather performance counter data.

Table 2 describes our testbed: an AMD Phenom 4-core CMP [1], with 2GB of memory, and a 200GB HDD that can independently clock each core at either 2.2 GHz or 1.1 GHz. Our machine lacks separate voltage planes: all cores use the same voltage. Given only two frequencies, we cannot test smaller frequency degradations or provide comprehensive statistical analyses of other frequency domains. We conservatively define a "bad core" as one that only runs at 1.1 GHz. Process variations might not always create such large discrepancies between cores, but this will identify points where the slower core clearly causes bottlenecks. Small frequency changes make it harder to tell whether faster cores are being bottlenecked due to the slower core, or whether they are constrained due to scaling factors such as parallelization overheads, thread spawning, and memory constraints. For example, a 16 core CMP with two cores operating at 20% less frequency might limit other cores, but this might not be apparent if cores are constrained by other factors such as off-chip memory or program scaling. Our baselines case is two cores clocked at maximum frequency, and our test cases are two fast cores and one slow core (at half frequency), and two fast cores and two slow cores.

We use a Watts Up Pro power meter to measure total wall-outlet power consumption. The meter's measurements are updated once per second. To isolate the processor power consumption, we measure power of the idle system clocked at the lowest frequency.

Table 1. NAS Benchmark Class B Inputs

APPLICATION	INPUT PARAMETERS
bt	102x102x102, 200 iterations
cg	75000, 75 iterations
ep	2147483648 Numbers
ft	512x 256x 256, 20 iterations
lu	102x102x102, 250 iterations
mg	256x 256x 256, 20 iterations
sp	2102x102x102, 400 iterations

Table 2. CMP Machine Configuration Parameters

Frequency	2.2 GHz (max), 1.1 GHz (min)
Process Technology	65nm SOI
Processor	AMD Phenom 9500 CMP
Number of Cores	4
L1 (Instruction) Size	64 KB 2-Way Set Associative
L1 (Data) Size	64KB 2-Way Set Associative
L2 Cache Size (Private)	512 KB/core 8-Way Set Associative
L3 Cache Size (Shared)	2 MB 32-Way Set Associative
Memory Controller	Integrated On-Chip
Memory Width	64-bits /channel
Memory Channels	2
Main Memory	2 GB PC2 5300(DDR2-667)

The processor consumes very little dynamic power in a low power state. We run our benchmarks from a networked file-server to minimize hard drive activity. We disable the OS's automatic dynamic voltage and frequency scaling, and the CMP runs on a single voltage domain for all processors. When determining power consumption of the CMP, we subtract idle power values. Idle power includes system power (motherboard, hard drive), memory and CPU at idle, with cores clocked to their lowest frequency. We find dynamic power consumption scales linearly with increasing threads for computation intensive benchmarks (*ep*). This confirms idle cores' power are not being masked by the active cores. All data logging is on a secondary machine.

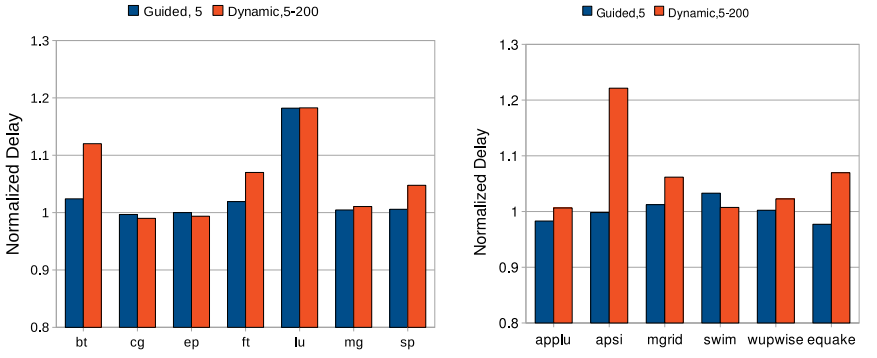
For the heterogeneous test case, we modify benchmarks to schedule threads at run-time. Our baseline uses pristine code without any modifications as that is most optimized for homogeneous processors with little scheduling overhead. Originally, the dynamic and guided scheduling constructs were for cases in which loop iterations took different amounts of time. Here, we use those scheduling constructs for load balancing between slower and faster threads.

5 Evaluation

We examine the performance, power, energy, and run-time scheduling characteristics for homogeneous workloads and compare with their heterogeneous counterparts at higher thread counts.

5.1 Performance

To gauge overhead of dynamic and guided versus static scheduling, we run NAS and SPEC OpenMP applications on three homogeneous cores and threads. Ideally, there should be no performance difference from dynamic load-balancing workload chunks among cores. Figure 2 shows delay increases from switching to dynamic or guided scheduling, with results normalized to our static baseline scheduler. *Guided,5* represents using guided scheduling with a minimum chunk size of five. *Dynamic,5-200* represents running all benchmarks using dynamic scheduling with fixed chunk sizes of 5, 10, 50, 100, 150 and 200 loop iterations, reporting the best performance from this range.



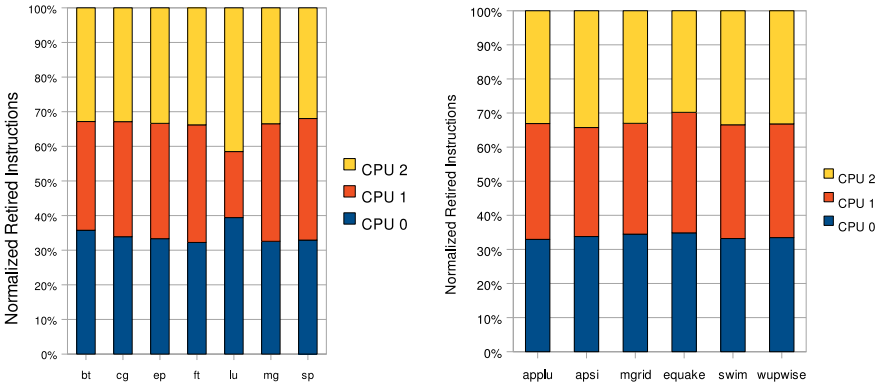
(a) NAS (b) SPEC-OMP

Fig. 2. Scheduling Performance Normalized to Default Static Scheduling (Lower is Better)

With NAS and SPEC benchmarks, changing dynamic chunk sizes significantly affects performance depending on the benchmark. There is almost no change in performance with the guided scheduler, except for *lu*, which exhibits an 18% increase in delay from dynamic scheduling versus static. This reduces benefits of dynamically allocating work for the third thread of this benchmark. Overall, we find guided to perform better than dynamic scheduling with chunk sizes, since the guided scheduler uses larger chunk sizes when possible. Guided only reduces the chunk size when backlogs occur. Even using a variety of chunk sizes for different programs, the dynamic scheduler performs worse for several benchmarks for which guided with just one lower bound value works well: *bt*, *ft*, *sp* and *apsi*. Small chunk sizes work well for codes that have large nested loops, while loops without nests benefit most from larger chunk sizes, as there is less memory contention between threads.

To gain insight into these performance differences, we examine how instructions are distributed across processors using the guided scheduling policy. Ideally, each core should equally receive a third of the total instructions. Figure 3 graphs instruction distribution across the homogeneous processors, where CPU 0, CPU 1, CPU 2 are the three cores on the CMP. All the SPEC-OMP benchmarks show excellent scaling, with about a third of the total instructions for each core, and only a 1% variation. The NAS benchmarks show more variation for two benchmarks. With *bt*, CPU 0 performs 5% more work than CPU 1 and 2% more work than CPU 3. *Lu* shows even higher variation, with CPU 0, CPU 1 and CPU 2 executing 39%, 19% and 42% of the total instructions respectively. This load imbalance leads to the performance degradation graphed in Figure 2(a), since static scheduling does not suffer from this phenomenon.

With perfectly scaling benchmarks that are computation bound (rather than memory bound), increasing from two to three threads should lead to a performance increase of 50%. However, since the third thread is only half as fast as the other two threads, only a 25% performance increase is expected. Since programs rarely scale perfectly and are not always computationally bound, this 25% improvement in performance is an expected upper limit of our results. Even the 25% upper bound may not be reached if the benchmarks are not partitioned equally at run-time between the cores.

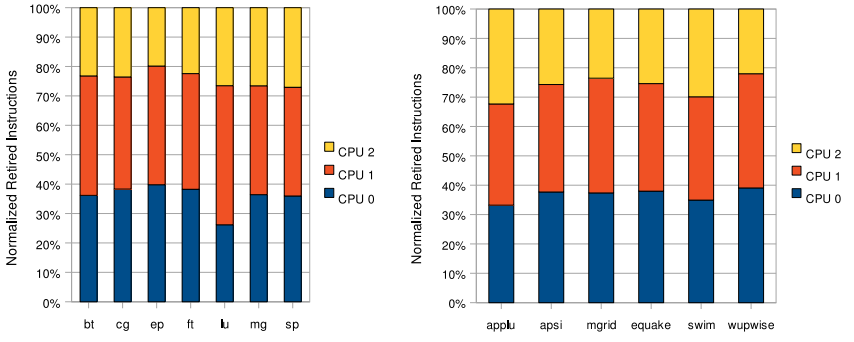


(a) NAS (b) SPEC-OMP

Fig. 3. Distribution of Instructions Across Homogenous cores

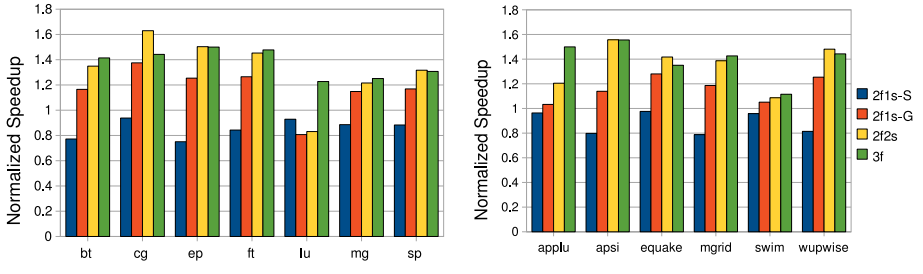
We examine the distribution of instructions across heterogeneous cores. Figure 4 graphs instructions across heterogeneous cores, where CPU 0 and CPU 1 operate at max frequency and CPU 2 runs at half the max frequency (1.1 GHz). Ideally, the fast cores should execute 40% of total instructions each, and the third core should process 20% due to the different operating frequencies between processors. Unfortunately, this perfect distribution only occurs for *ep*. On average, the slower core actually processes 25% of total instructions. This is 5% more total chip-wide instructions than the ideal case, and 25% more for that one core. This imbalance reduces the efficiency of the heterogeneous configuration. The load imbalance is worst with *swim* and *applu* where CPU 2 executes 30% and 32% of the total instructions respectively. This does not degrade performance for *swim* significantly since it is already memory bound at the higher thread count. *Swim* has the largest memory footprint of all the SPEC benchmarks and traditionally does not scale well with threads going from two to four threads [2]. *Lu* displays a different form of imbalance between the two homogeneous cores. For most benchmarks the homogeneous cores distribute the remaining workload equally, but with *lu* CPU 1 executes 47%, and CPU 0 and CPU 2 execute 26% and 27% respectively.

Speedups of running benchmarks on the heterogeneous three core system are graphed in Figure 5(a) for NAS and Figure 5(b) for SPEC OMP. Results illustrate four different cases normalized to the two processor high frequency architecture baseline. *2f1s-S* is the heterogeneous CMP (composed of two fast cores and one slow core) with static scheduling for thread workloads. *2f1s-G* is the guided scheduling of thread workloads on a heterogeneous three core CMP, and *2f2s-G* is the guided scheduling of thread workloads on a heterogeneous four core CMP (composed of two fast cores and two slow cores). *3f* is the three thread homogeneous case where all processors run at the maximum frequency. Results are normalized to their two thread counterparts. *3f* and *2f2s* perform similarly, with *3f* doing better for benchmarks with workload imbalances (applu, lu) and *2f2s* doing better for benchmarks whose cache footprint decreases with increasing threads (cg). *2f1s-G* performance is between *2f1s-S* and *3f*. For memory



(a) NAS (b) SPEC-OMP

Fig. 4. Distribution of Instructions Across Heterogenous cores



(a) NAS (b) SPEC-OMP

Fig. 5. Performance Normalized to Two-Thread CMP (Higher is Better)

constrained benchmarks, *2f1s-G* performs competitively with *3f*. For computation limited codes, *2f1s-S* performs poorly, since the slower core bottlenecks the faster ones, resulting in an effective raw clock speed of 3.3GHz compared to the baseline's 4.4 GHz (25% slower). Performance degradation varies with benchmark, depending on thread barriers and synchronization points. *2f1s-S* is not included in further results, since its results are obvious and offer little insight into desirable power and performance. The NAS benchmarks with the *2f1s* setup benefit significantly from the increase in threads. Although the extra processor runs at half the frequency of the other cores, effective workload balancing ensures it does not throttle performance at synchronization points. The *2f2s* configuration also is able to perform as well as the *3f* configuration, effectively compensating for the one less fast core with two slow ones.

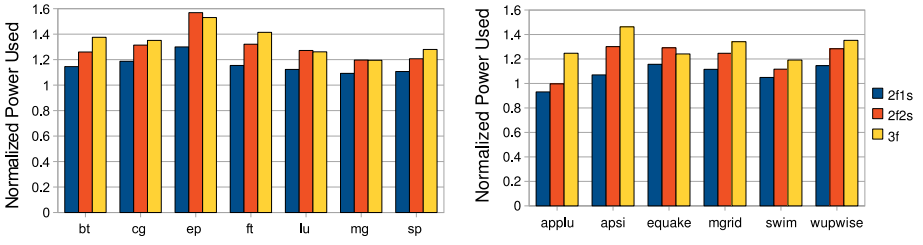
The exception to our positive performance gains is *lu*. Its performance degrades on heterogenous CMP configurations, by 19% (for *2f1s-G*) and 17% (for *2f2s*). There are several reasons for this degradation. Faster cores have to wait for the slower core at barrier points, leading to the faster cores being idle. Another is the synchronization overhead of dynamic chunks sizing, which can slow the system significantly. Figure 2 confirms our previous observations of reduced performance from dynamic scheduling overhead. Figure 4 confirms the workload imbalance between cores.

2f1s-G's *cg* benchmark shows the most gains of the NAS benchmarks, with a 38% super-linear gain in performance. *Cg*'s performance gains are due to decreasing cache misses with increasing threads. From the SPEC suite, *2f1s*'s *equake* shows the most gains, also with a slight super-linear improvement in performance of 28%. *Swim* only exhibits a 5% speedup with *2f1s-G*, and even *3f* only improves performance by 12%, which indicates the processors are often idle waiting for memory. Running with more threads results in increased off-chip memory pressure, which is why *swim* fails to improve in performance. We next examine average per-benchmark power to confirm this.

5.2 Power and Energy

Activating a third processor at the same voltage but lower frequency yields less energy savings than lowering voltage in tandem with frequency. This is due to the quadratic coefficient voltage plays in the power equation($\frac{1}{2}CV^2f$). There should be a 25% net increase in CPU power consumption for computation intensive benchmarks. Actual energy consumption varies since dynamic power consumption for various processes (L3 cache accesses, memory controller, main memory functions, off-chip accesses, and cache coherence) are amortized across all cores.

Figure 6 graphs the power consumption of the three processor *2f1s* and *3f*, and four processor *2f2s* cases, normalized to the two processor scenario. *2f1s*'s power consumption is about half way between the power consumption of the *2f2s* and baseline configuration for most benchmarks. The *3f* configuration generally has the highest power consumption. A workload imbalance reduces throughput, but does not significantly reduce power consumption of the cores for the *2f2s* and *2f1s* configurations. Although *lu* shows performance degradation for *2f1s* and *2f2s*, power consumption increases by 12% and 27% respectively. *Ep* power consumption with the *2f1s* configuration is 30% higher than the two thread case, slightly higher than the theoretical increase of 25% for computation bound applications. The higher consumption can be attributed to increased power consumption of shared structures, which operate at a rate independent of the slow core, such as cache coherence. *applu* exhibits a slight increase in performance (3%) and decrease in power power consumption (7%) with the *2f1s* configuration. The slower thread for *2f1s* slows down the others, such that increasing number of threads does not improve performance, and reduces power consumption of the faster running processors. This is proven from the instruction mix and performance graphs seen earlier. *applu* exhibits a 20% improvement in performance with no increase in power with the *2f2s* configuration. This would not be noteworthy if we were reducing frequency and voltage when increasing threads. However, here we increase the number of threads without reducing the frequency or voltage from cases using threads. This indicates that for the two-thread configuration, the threads are performing inefficiently. For *lu*, *2f1s* and *2f2s* are also inefficient configurations, where power increases by 12% and 27%, while performance actually degrades. This is because processors are still spinning on locks, executing useless instructions while stalled at barrier points. Overall, *2f1s* and *2f2s* use 12% and 26% more power respectively on average, during execution. This is lower than our theoretical maximum, but accurately matches the trends in performance improvement seen above. The exception to this being *lu*.

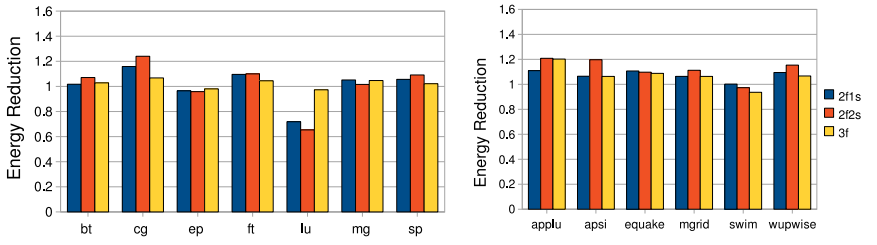


(a) NAS (b) SPEC-OMP

Fig. 6. Power Consumption Normalized to Two Thread CMP (Lower is Better)

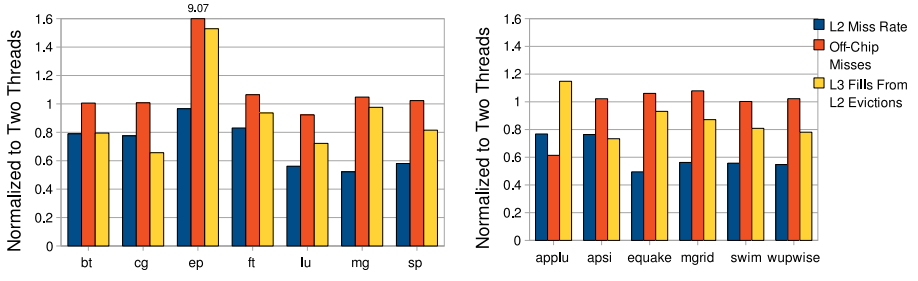
Using more threads requires more power, since voltage or frequency do not decrease. However, if the benchmarks achieve a significant performance improvement, total energy is decreased due to power being consumed for less time. This is because shared structures are active for less time, and whose dynamic power consumption is amortized over greater threads. We compare total energy consumption of the *2f1s*, *2f2s*, and *3f* configurations with the two-core homogeneous CMP. Figure 7(a) graphs energy consumption for NAS, while Figure 7(b) shows energy consumption for SPEC. Results are normalized to the baseline homogeneous case. To attain these energy savings, we assume the system can enter a low power state (be turned off) or start on the next job, when idle after completing the previous job. With the NAS benchmarks, energy reduction varies, depending on performance improvements and increased power consumption. *Ep* requires slightly more energy with more threads, since the improvements in performance closely correlate with increases in power consumption. *Cg, ft* and *sp* show highest energy reductions with the heterogeneous configurations. For these benchmarks, improvements from increasing threads, and improvements from reduced cache misses leads to greater efficiency. This is especially true for *cg*, which is why it has the highest energy reductions with more threads. *Lu* shows significant increases in energy consumption, due to the decrease in performance, and increase in power consumption from using heterogenous configurations. For *lu*, the homogenous configuration does not improve over the baseline, since improvements in performance are offset by similar increases in power consumption.

All SPEC benchmarks except *swim* show energy reductions for all configurations. Recall that *swim* is memory constrained and suffers from many memory stalls, thus running extra threads without decreasing voltages or frequencies fails to improve energy efficiency. The worst degradation is less than 6%, however: while processors are stalled waiting for memory, their power consumption is low. *3f* is the most inefficient for *swim*, another indication that high frequencies do not help. From 7(b), its clear the heterogenous configurations are just as or more energy-efficient than the homogenous configurations. The exception being applu, where workload imbalances hampers energy reductions, and efficient operation. When instructions are stalled due to memory latency, processors attempt to issue other instructions by reordering instructions to execute. This leads to extra work that might not be done at lower frequencies, where memory from the processor's perspective has a lower latency. This is one of the reasons memory bound benchmarks have higher energy savings with the *2f1s* configuration compared to the *3f* configuration.



(a) NAS (b) SPEC-OMP

Fig. 7. CMP Energy Reduction Normalized to Two Thread Configuration (Higher is Better)



(a) NAS (b) SPEC-OMP

Fig. 8. *2f2s* vs. *2f* Cache Behavior (Lower is Better)

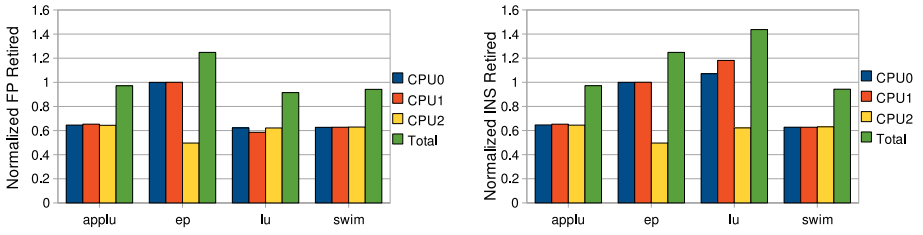
We examine cache activity from switching a homogenous two-thread configuration to a heterogenous four-thread configuration. Figure 8 shows memory behavior for the four thread *2f2s* case normalized to the two thread *2f* case. Individual L2 cache miss rates, total off-chip misses and L3 cache fills from L2 evictions are graphed. Reductions in L2 cache misses improve performance and reduce energy consumption. Since off-chip misses do not increase with increasing threads, the off-chip power is amortized over more cores, so efficiency (defined as performance per watt) increases. The reductions in L3 fills from L2 evictions reduces L3 cache accesses, reducing energy consumption. Benchmarks exhibit reductions in L2 miss rates and L3 fills from reduced L2 misses, with increasing threads. *Ep* has significant increases in off-chip misses (900%) with increasing threads. However, actual values are very small, with a negligible effect on performance. While these reductions improve performance and energy, their effect on energy reduction will vary, depending on the ratio of microarchitecture activity to cache accesses. We do not graph L2 cache coherence snoops, which should increase with increasing threads due to our broadcast based cache coherence. However, L2 snoops only require checking the cache tags, requiring significantly less energy than a cache miss that entails line fills and off-chip accesses.

A user might be inclined to clock down one of the cores on a CMP and inherit this "heterogeneous" configuration, such as with *2f1s*, trading off performance for power and energy reductions. Alternatively, one can tradeoff die area for energy reductions with the *2f2s* configuration.

5.3 Metrics for Run-Time Scheduling

We previously found increasing threads does not always improve performance, and even when it does, it might not be energy efficient. It would be useful to determine the optimal number of threads in real-time without offline profiling for frequency-heterogeneous CMPs. To achieve this, we need to be able to determine thread progress, and application power consumption. We examine using performance counters to determine useful thread progress, and whether increasing thread counts improves performance. We use power meters to determine real-time power consumption of the application for a given configuration. We specifically examine total instructions, and total floating-point (FP) instructions retired (inclusive of MMX instructions) per processor to guage thread progress. We examine whether using either of these performance counters is sufficient to determine thread progress. The drawback of the total instructions retired metric, is that stalled threads spinning on locks can lead to increased instruction counts without increasing work throughput. Unfortunately, OpenMP does not support dynamically changing the number of threads via signals sent outside the program, which prevents us from implementing a real-time scheduler that detects whether increasing the thread count reduces performance. Our results are thus based on static thread counts, but nothing precludes this work from being done in real-time, should the underlying shared-memory framework support it.

Figure 9(a) graphs floating-point instructions per core per second for *applu*, *swim*, *lu*, and *ep* for the *2f1s* configuration, normalized to the two thread homogeneous CMP scenario. We choose these benchmarks since they suffer the worst degradation from increasing thread counts, except for *ep*, which we use as a point of comparison. For the third core, we normalize retired FP instructions to one of the homogeneous cores (which of the two cores we choose does not matter, since both retire approximately the same number of FP instructions) from the two-thread runs. The *total* column represents total FP throughput normalized to the homogeneous two-thread case. Ideally, for computation bound cases there should be no change in cores 0 and 1 when thread counts increase, and 50% throughput for core 2. This is the case for *ep*, which is why it has a 25% increase in total FP instructions retired. Other benchmarks show large degradations for cores 0 and 1 when thread counts increase, resulting in fewer total FP instructions retired than for the two thread case. Optimal thread counts can thus be chosen dynamically (without profiling). Interestingly, core 2 shows throughput higher than 50% for many benchmarks, indicating they are bound by memory and not by frequency. We do not use total instructions committed to gauge throughput, since all of the benchmarks we evaluate are FP, and processors spinning on locks may increase integer throughput without improving performance. This phenomenon is illustrated for *lu* in Figure 9(b), which graphs the total instructions per core per second. Results are normalized to the two-thread homogeneous CMP scenario. While other benchmarks follow the trends from Figure 9(a), *lu* shows how using the total instruction metric could lead to erroneous thread choices since it exhibits higher instruction throughput at higher thread counts. Processor stalls are not a good metric to detect stalls for applications limited by memory or barriers such as *lu*, since cores are not stalled when spinning on locks. These results can be used for sampling the entire search space at run-time until the optimal configuration is found.



(a) Floating-Point (FP) (b) Integer (INT)

Fig. 9. CMP Retired Instructions Scaling from Two to Three Threads

6 Conclusions

We achieve performance and energy improvements by leveraging cores that operate at different frequencies on real hardware. By balancing workload chunks between faster and slower cores, we reduce bottlenecks caused by slow cores. We find dynamic scheduling methods can add noticeable overheads and affect performance. The guided scheduling method with a single lower bound for chunk size works well for most benchmarks, while dynamic scheduling with fixed chunk sizes does not. Workload balancing is imperfect, with a variation of 5% above ideal on average. With three heterogeneous cores, this yields a 25% load imbalance on the slowest core.

We observe speedups of up to 38% and reduce total CPU energy consumption by up to 16%. While increasing numbers of threads increases power consumption, reductions in execution time yield overall reductions in energy consumption. We find running memory-limited benchmarks in heterogeneous configurations to be competitive with homogeneous configurations at identical thread counts, at greater energy reductions.

We describe a feasible method to detect at run time whether increasing thread counts will improve performance. We would like to extend our work to determine whether using extra cores is more energy efficient. We will expand our study to include scheduling frequency heterogeneous CMPs with separate voltage domains per processor. As the number of on-chip cores grows, significant inter-core heterogeneity will be inherent. Managing this will be crucial in determining the future processor landscape.

References

1. AMD Corporation. Model number and feature comparisons — AMD Phenom. Processors, `http://www.amd.com/us-en/Processors/ProductInformation/030_118_15331_15332`
2. Aslot, V., Eigenmann, R.: Performance characteristics of the SPEC OMP2001 benchmarks. In: Proceedings of the European Workshop on OpenMP (September 2001)
3. Bailey, D., Harris, T., Saphir, W., Van der Wijngaart, R., Woo, A., Yarrow, M.: The NAS parallel benchmarks 2.0. Report NAS-95-020, NASA Ames Research Center (December 1995)
4. Balakrishnan, S., Rajwar, R., Upton, M., Lai, K.: The impact of performance asymmetry in emerging multicore architectures. In: Proc. 32nd IEEE/ACM International Symposium on Computer Architecture, pp. 506–517 (June 2005)

5. Bowman, K., Alameldeen, A., Srinivasan, S., Wilkerson, C.: Impact of die-to-die and within-die parameter variations on the throughput distribution of multi-core processors. In: Proc. IEEE/ACM International Symposium on Low Power Electronics and Design, pp. 50–55 (August 2007)
6. Bowman, K., Meindl, J.: Impact of within-die parameter fluctuations on future maximum clock frequency distributions. In: Proc. IEEE Conference on Custom Integrated Circuits, pp. 229–232 (May 2001)
7. Donald, J., Martonosi, M.: Power efficiency for variation-tolerant multicore processors. In: Proc. IEEE/ACM International Symposium on Low Power Electronics and Design, pp. 304–309 (October 2006)
8. Eranian, S.: Perfmon2: a flexible performance monitoring interface for Linux. In: Proc. 2006 Ottawa Linux Symposium, pp. 269–288 (July 2006)
9. Ge, R., Feng, X., Feng, W., Cameron, K.: CPU MISER: A performance-directed, run-time system for power-aware clusters. In: Proc. International Conference on Parallel Processing, pp. 18–26 (September 2007)
10. Herbert, S., Marculescu, D.: Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In: Proc. IEEE/ACM International Symposium on Low Power Electronics and Design, pp. 38–43 (August 2007)
11. Humenay, E., Tarjan, D., Skadron, K.: Impact of process variations on multicore performance symmetry. In: Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition, pp. 1653–1658 (August 2007)
12. Isci, C., Buyuktosunoglu, A., Cher, C.-Y., Bose, P., Martonosi, M.: An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In: Proc. IEEE/ACM 40th Annual International Symposium on Microarchitecture, pp. 347–358 (December 2006)
13. Isci, C., Contreras, G., Martonosi, M.: Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In: Proc. IEEE/ACM 40th Annual International Symposium on Microarchitecture, pp. 359–370 (December 2006)
14. Kadayif, I., Kandemir, M., Vijaykrishnan, N., Irwin, M., Kolcu, I.: Exploiting processor workload heterogeneity for reducing energy consumption in chip multiprocessors. In: Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition, vol. 2, pp. 1158–1163 (February 2004)
15. Liao, C., Liu, Z., Huang, L., Chapman, B.: Evaluating OpenMP on chip multithreading platforms. In: Proc. First International Workshop on OpenMP (June 2005)
16. Liu, F., Chaudhary, V.: Extending OpenMP for heterogeneous chip multiprocessors. In: Proc. International Conference on Parallel Processing, pp. 161–170 (October 2003)
17. Standard Performance Evaluation Corporation. SPEC OMP benchmark suite (2001), `http://www.specbench.org/hpg/omp2001/`
18. Tiwari, A., Sarangi, S., Torrellas, J.: ReCycle: Pipeline adaptation to tolerate process variation. In: Proc. 34th IEEE/ACM International Symposium on Computer Architecture, pp. 323–334 (June 2007)
19. Tiwari, A., Torrellas, J.: An updated evaluation of ReCycle. In: Proc. Workshop on Duplication, Deconstructing, and Debunking, in association with the 35th International Symposium on Computer Architecture (June 2008)
20. Unsal, O., Tschanz, J.W., Bowman, K., De, V., Vera, X., Gonzalez, A., Ergin, O.: Impact of parameter variations on circuits and microarchitecture. In: Proc. IEEE/ACM 40th Annual International Symposium on Microarchitecture, pp. 30–39 (November 2006)
21. Wong, P., Jin, H., Becker, J.: Load balancing multi-zone applications on a heterogeneous cluster with multi-level parallelism. In: Proc. of the Third International Symposium on Parallel and Distributed Computing, pp. 388–393 (July 2004)

A Framework for Task Scheduling and Memory Partitioning for Multi-Processor System-on-Chip

Hassan Salamy and J. Ramanujam

Department of Electrical and Computer Engineering
and Center for Computation and Technology
Louisiana State University, Baton Rouge, LA 70803, USA
{hsalam1,jxr}@ece.lsu.edu

Abstract. The growing trend in current complex embedded systems is the use of multiprocessor system-on-chip (MPSoC). An MPSoC consists of multiple heterogeneous processing elements, a memory hierarchy, and input/output components which are linked together by an on-chip interconnect structure. Using such an architecture provides the flexibility to meet the performance requirements of multimedia applications while respecting the constraints on memory, cost, size, time and power. Such embedded systems employ software-managed memories known as *scratch-pad memories* (SPM). Scratchpad memories, unlike caches, are software-controlled and hence the execution time of applications on such systems can be accurately predicted and controlled. Scheduling the tasks of an application on the processors as well as partitioning the available SPM budget among those processors are two critical issues in reducing the overall computation time as well as the communication overhead. Traditionally, the step of task scheduling is applied separately from the memory partitioning step. Such a decoupled approach may miss better quality schedules. In this paper, we present an effective heuristic that integrates task allocation and SPM partitioning to further reduce the execution time of embedded applications. Results on several real life benchmarks show the significant improvement of our proposed technique compared to decoupled techniques as well as to an integer-linear programming approach.

1 Introduction

Thanks to recent advances in architecture, VLSI and electronic design, the current trend in modern complex embedded system design is to deploy a multiprocessor system-on-chip (MPSoC). Generally speaking, an MPSoC consists of multiple heterogeneous processing elements (PEs), memory hierarchies, and I/O components interconnected by complex communication architectures. Such architectures provide the flexibility of simple design, high performance, and optimized energy consumption. An MPSoC provides an attractive solution to the problems brought forth by increasing complexity and size of embedded systems applications. Execution time predictability is a critical issue for real-time embedded applications; this makes the use of data caches not suitable as a cache

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 263–277, 2009.

is hardware-controlled and hence it is hard to model the exact behavior and to predict the execution time of programs. To alleviate such problems, many modern MPSoC systems use software-controlled memories known as *scratch-pad memories* (SPM).

An SPM is fully software-controlled and hence the execution time of an application on such memories can be predicted with accuracy. Unfortunately, scratch pad memories are expensive and hence they are usually of limited size and as a result not all the application data variables can be stored in the on-chip scratch-pads. Many multi-processor system-on-chip models use a memory hierarchy with slow off-chip memory (DRAM) and fast on-chip scratchpad memories. Such a hierarchy means that proper allocation of variables to the on-chip memory is an essential part in reducing the off-chip accesses. The computation time of a program on a processor depends on how much SPM is allocated to that processor as accessing an element from the off-chip memory is usually in the order of 100 times slower than accessing elements stored locally in the on-chip memory.

An embedded application can usually be divided into multiple tasks, and different tasks can be scheduled on different processors. The computation time for each task depends on the amount of SPM allocated to the processor executing this task. The problem of task scheduling and memory allocation on MPSoCs is an NP-complete problem [12]. Traditionally, these two steps are done separately where tasks are usually scheduled and the SPM budget is then partitioned among the processors. Such a decoupled technique may not result in better schedules in terms of minimizing the computation time of the whole application. The appropriate configuration of a processor's scratch pad memory depends on the tasks scheduled on that processor. Therefore, the integration of those two steps is critical to improve the performance. In this paper, we present a heuristic that performs task scheduling and SPM memory partitioning in an integrated fashion where the private on-chip memory budget allocated to a processor is decided dynamically as tasks are mapped to this processor.

The remainder of this paper is organized as follows. Section 2 presents the problem definition, motivation and the heuristic for integrated task scheduling and memory partitioning. Section 3 presents an example to further clarify our technique. Section 4 presents experimental results. Section 5 presents related work in this area. Finally, Section 6 presents our conclusions.

2 Task Scheduling and Memory Partitioning

2.1 Architecture Overview and Problem Definition

Dividing an application into a set of tasks where one or more independent tasks can be executed in parallel on the available processors is extremely useful for MPSoCs. Parallelism leads to potential for speeding up the execution time; this is a major issue in embedded processors. A typical MPSoC is shown in Figure 1 which consists of multiple processors, an SPM budget divided among the processors, and a global off-chip memory that can be accessed by all the processors.

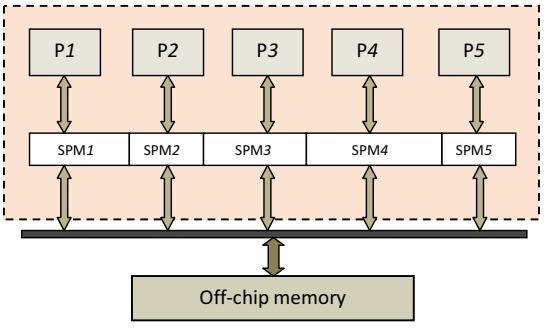


Fig. 1. An architectural model example with 5 processors, SPM budget, off-chip memory and interconnection buses

Our problem formulation is based on a task dependence graph (TDG). A TDG is a directed acyclic graph with weighted edges where each vertex represents a task in the embedded application. An edge between two tasks, say T_i and T_j in the TDG, represents some kind of a scheduling order that needs to be enforced due to the fact that T_j needs data to be transferred from T_i after T_i is already executed. A certain processor cannot start executing task T_j unless all the necessary data communication is performed. The weight of an edge is the communication cost. Each task can be mapped to any of the available processors. Since the processors in our architectural model can be heterogeneous, the execution time of each task depends on the processor this task is mapped to as well as the SPM memory allocated to that processor. Generally speaking, a larger SPM results in less computation time since off-chip access is more expensive in terms of the clock cycles compared to fast on-chip SPM. A large portion of the execution cycles of a task goes to accessing the data variables. Accessing a data variable from an SPM is usually in the order of 100 times faster than accessing it from the off-chip memory. Since the available SPM memory is usually limited due to the multiprocessors design constraints, a good utilization of SPM can be critical in narrowing the gap with the processor's speed.

The problem can now be stated as follows: Given an embedded application consisting of t tasks, an MPSoC architectural model and an SPM budget: (i) find a schedule of those tasks on the available processors, (ii) partition the SPM memory among the processors, and (iii) assign data variables of a certain task T scheduled on processor P to the private SPM budget assigned to P. The objective is to minimize the execution time in cycles of the embedded application on the MPSoC architectural model.

2.2 Motivation

Most works so far have treated task scheduling and memory partitioning as two decoupled steps that are performed independently. Given a set of tasks and an MPSoC model with a certain amount of available scratch pad memory budget,

tasks are usually scheduled on the processors and then memory is partitioned among used processors. In this aspect, those two steps are performed independently. However, the configuration of a processor's scratch pad memory is highly dependent on the tasks scheduled on this processor. Thus, task scheduling and memory partitioning are inter-dependent on each other and they should be integrated in one step in order to get high quality schedules.

The computation time of a task depends on the processor it is mapped to as well as on the SPM memory available for that task. Therefore, task scheduling should take into consideration the varying computation time of a task based on the processor and on the SPM budget. Considering static computation time, meaning that the computation time is fixed from the scheduler point of view, may limit the quality of the schedule.

Consider the example in Figure 2(a) of a task graph with 6 tasks, *T1, T2, T3, T4, T5,* and *T6*. Task *T4* depends on tasks *T1, T2* and *T3,* and task *T6* depends on tasks *T4* and *T5*. Anytime there is an edge between two tasks *Ti* and *Tj* means that a communication cost should be accounted for provided that those two tasks are allocated to two different processors. Although our technique takes such costs into account, we omit them in Figure 2 for simplicity. Define the Min_{ij}, Avg_{ij}, and Max_{ij} as the computation time for task T_i on processor P_j assuming all of the available SPM budget is assigned to P_j, $1/n$ of the available SPM budget is assigned to P_j where n is the number of processors, and no SPM respectively. Those values will be used later on by our heuristic. In this example, we assume two homogeneous processors. The (Min, Avg, Max) values are shown in Figure 2(a). Figure 2(b) shows the schedule assuming no available scratch pad memories. First tasks *T1* and *T2* will be mapped to the two available processor $P1$ and $P2$. At this time only task *T3* is ready to be scheduled. The scheduling algorithm will map *T3* to $P2$ as it is free before $P1$ since the computation time of *T2* is less than that of *T1*. In a similar fashion, the scheduling algorithm will assign tasks *T4* and *T6* to processor $P1$ whereas task *T5* will be mapped to processor $P2$. The cost of such a schedule is equals to 29.

Figure 2(c) shows the results following the common practice of partitioning the available SPM memory equally between the two processors. With such a criterion, the available SPM budget will be equally divided between processors $P1$ and $P2$ regardless of what tasks are mapped to what processors. Equally partitioned SPM reduces the computation time of the whole application to 25.

To further reduce this application's computation time, the available SPM can be divided between the two processors in any ratio. From the task schedule, we can see that task $T4$ can start only after $P2$ is done executing task $T3$. The issue now is to try to reduce the dead time between tasks $T1$ and $T4$ imposed by the computation time for tasks $T2$ and $T3$. To minimize this dead time, techniques usually allocate more SPM budget to processor $P2$ to reduce the computation time of tasks $T2$ and $T3$. Notice that if all the SPM memory is allocated to processor $P2$ then the computation time for $T1$ will jump to 15 and as the results the minimum start time of $T4$ will increase from 14 to 15. To avoid this increase, some SPM memory should be allocated to $P1$ to keep the execution

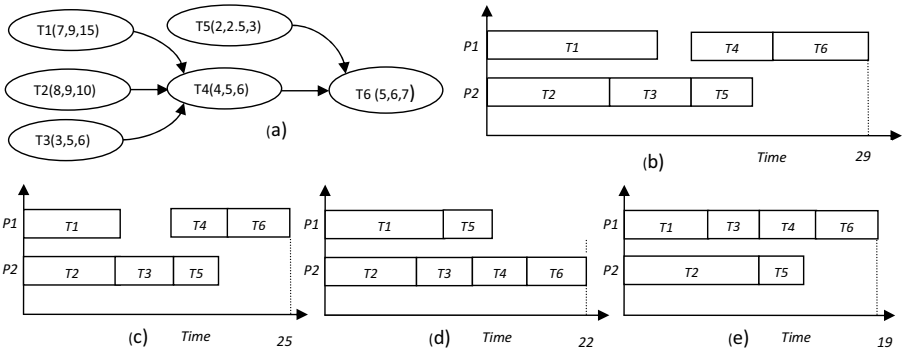


Fig. 2. (a) TDG. Schedule based on: (b) no SPM. (c) equal partitioned SPM. (d) non equal partitioned SPM. (e) our integrated approach.

time as balanced to the end time of $T3$ as possible. Intuitively speaking, the approximated minimum end time of $T3$ will be 12 and thus the total computation time for our example application will be close to 23. With the same memory partitioning, the computation time can be reduced to 22 assuming that tasks $T4$ and $T6$ are scheduled on P_2 and task $T5$ is mapped to P_1; Figure 2(d).

However, 22 is not the optimal time for scheduling the example task graph on two processors. Our heuristic, presented later, can reduce the computation time to 19 as it integrates task scheduling and memory allocation into one step. The problem with the previous schedule is that it allocated T_3 to the same processor P_2 that is scheduled to execute T_2. This choice is the reason for the dead time in the schedule as T_2 cannot benefit much from more SPM memory which is clear from the Min, Avg, and Max values. A good heuristic should take those values into consideration where a better choice for T_3 is to be scheduled on P_1 with all available SPM memory is allocated to this processor and the results is a schedule with minimal end time of 19; see Figure 2(e). A benchmark example is presented in Section 3.

2.3 Our Heuristic

A good heuristic for task scheduling and memory partitioning should take into consideration the dynamic (varying) execution time of a task throughout the process of building the schedule. This dynamic execution time is the result of the dynamic SPM budget assignment to processors throughout the course of the heuristic. Using profiling of the tasks in the embedded application, *Min, Avg,* and *Max* values (defined earlier) are calculated for each task on each of the available heterogeneous processors. We define *elasticity* of a task as the extent to which this task can benefit from a larger SPM. Although it can be defined in different ways, we define *elasticity* dynamically as the extent to which the computation cost of a task on P_i may decrease as the SPM budget of P_i is increased from the current budget to *size* where *size* is the maximum amount of SPM budget available in our model. Equation 1 defines *elasticity* of task T_i where *Cur* is the

computation time of the task under the current memory budget. The *elasticity* of a task T_i is basically a measure of the room for computation time reduction of T_i with more SPM budget.

$$elasticity(T_i) = \frac{Cur_i - Min_i}{Cur_i} \quad (1)$$

A bigger value of *elasticity* means that the computation time of T_i is more amenable for reduction with the increase in the SPM allocated to that task. Note that $elasticity(T_i)$ is a dynamic value since the current computation time of T_i, Cur_i, may change as the SPM budget distribution changes.

Our heuristic in Figure 3 starts with profiling the application to extract important information. Using the profiling data, the embedded application will be divided into tasks with a necessary data communication between two tasks impose a certain kind of dependency. Based on the extracted tasks and communication between them, the task dependence graph is created. In this graph, each task is represented by a vertex and each communication cost by a weighted directed edge. For each available task T_i and processor P_j, we calculate the number of variables, the size of the variables, Min_{ij}, Avg_{ij}, and Max_{ij} values. All those values are computed through profiling. Then the ASAP values for all tasks are calculated based on the Avg values that is assuming the SPM budget is equally divided among the available processors. Tasks will be sorted in increasing order of the ASAP values in a list $L1$. For each task, following the ASAP sort, we evaluate the best processor to assign this task to so that the overall computation time is minimally increased.

The minimum start time of a task T_i on processor P_j, $Start\_time(T_i, P_j)$, is equal to the maximum of the end time of processor P_j, $End\_time(P_j)$ and the maximum end time of all its parent tasks, $Max_{T_j \in Parent(T_i)}(T_j)$, plus the maximum communication time of all the parent tasks scheduled on P_k with $k \neq j$ (see Equation 5). Two dependent tasks mapped to the same processor will have zero communication cost. In general, task T_i will be scheduled on the processor P_j corresponding to the minimum additional overhead time in the schedule. However, T_i may be scheduled on a processor P_l of higher overhead time provided that the predicted end computation time ($PEC(P_l)$) (defined by us in Equation 2) of this processor is at least δ % less than that of P_j. We choose δ of 10 in our experimental evaluations. This $PEC(P_l)$ value is a guide to the scheduler of how much this over head time may decrease with the SPM memory transfers in future steps if T_i is mapped to P_l. PEC is basically an estimate of how much the end time of processor P_l will be if more SPM budget is assigned to it. The PEC of a processor is closely related to the *elasticity* of the tasks scheduled on that processor. The PEC value provides the dynamic essence of our heuristic as at each step the heuristic looks beyond the current SPM budgets distribution in its task mapping decision to an estimate of future distribution in future steps. In the case of equal additional end time, if task T_i is assigned to two different processor then we avoid assigning it to a processor with no scheduled tasks. In this case, we schedule T_i on the processor with the

higher *elasticity* under the current SPM budget. The *elasticity* of a processor is the average value of the *elasticity* of the tasks scheduled on this processor.

$$PEC(P_i) = End\_time(P_i) - \sum_{T_j \in P_i} \left(Cur(T_j) - \frac{Cur(T_j)}{1 + elasticity(T_j)} \right) \quad (2)$$

After scheduling any task, we try to balance the schedule in a way to decrease the total computation time. We do so by dynamically changing the SPM budget for each processor to reach a better balance. We start by trying to reduce the computation time of tasks on processor P_i with maximum end time so far. We do so by transferring an α % of the memory budget, Mem_j, corresponding to processor P_j with the minimum (End_time*$elasticity$) and such that $End\_time(P_j) < End\_time(P_i)$ and assigning it to processor P_i. Doing so will probably decrease the end time of processor P_i and in the same time increase the end time of processor P_j. Considering processor P_j to be of low total *elasticity* will give more room to reduce its SPM budget with a minimal increase in its End_time. We do memory transfer α% at a time as long as $End\_time(P_j) < End\_time(P_i)$.

$$Time(T_i, Mem_j) = Time(T_i, 0) - Gain(T_i, Mem_j) \quad (3)$$

$$Gain(T_i, Mem_j) = \sum_{v_i \in T_i, v_i \in Mem_j} ((\beta_1 - \beta_2) * freq_i). \quad (4)$$

$$\begin{aligned} Start\_time\,(T_i, P_j) = & Max\left(Max\left(End\_time_{T_k \in Parent(T_i)}\,(T_k)\right), End\_time\,(P_j)\right) \\ & + \; Max\left(Comm\_time_{T_k \in Parent(T_i)}\,(T_k)\right) \end{aligned} \quad (5)$$

$$End\_time(T_i) = Start\_time(T_i, P_j) + Time(T_i, Mem_j) \quad (6)$$

$$End\_time(P_j) = Max(End\_time_{T_k \in P_j}(T_k)) \quad (7)$$

After any SPM memory budget redistribution among different processors, the *Recompute*() subroutine will be invoked to recompute the start time, computation time, and end time of tasks T_i referred to respectively as $Start\_time(T_i)$, $End\_time(T_i)$, and $Time(T_i)$. First a *Gain* value, $Gain(T_i, Mem_j)$, is computed for T_i with the newly budget SPM memory assigned to the processor T_i is mapped to. This *Gain* value in Equation 4, represents the execution cycles reduced due to allocating variables of T_i to Mem_j following the increasing order $byte/freq$ of the data variables where $byte_i$ is the size of the variable v_i and $freq_i$ is the number of times v_i is accessed. In Equation 4, β_1 is the cost of accessing a variable from the off-chip memory and β_2 is the cost of the SPM access. This is a simple data allocation technique that we adopted in our heuristic. The new computation time of T_i, $Time(T_i, Mem_j)$ is the time taken to execute T_i assuming no SPM memory, $Time(T_i, 0)$, minus $Gain(T_i, Mem_j)$. We assume on-chip memory access costs only one clock cycle. The end time of a task T_i scheduled on processor P_j is then calculated as in Equation 6. The end time of each processor is thus the end time of the last task assigned to this processor (Equation 7).

After all the tasks are scheduled, we call the *Balance*() procedure to try to further reduce the schedule cost through reducing the end time of the processor

with the largest end time through memory transfer. At this point we tune the $Balance()$ procedure so that it allows the last memory transfer between P_i and P_j that will result in $End\_time(P_j) > End\_time(P_i)$. We run this procedure t times where t is the number of tasks in the TDG. Notice that if a processor ends up with no scheduled tasks, then the SPM budget for such processor will be distributed among other processors using the $Balance()$ procedure to reduce the schedule time the most.

2.4 Pipeline Scheduling

An embedded application is usually executed many times for a stream of input data on a MPSoC. Such multiple executions make the embedded application amenable to pipelined implementation. Pipeline scheduling benefits from allowing tasks from different embedded application instances to be scheduled at each stage of the pipeline. Such a schedule does not necessarily decrease the computation time of one instance of embedded application but rather it decreases the time between the start of two consecutive iterations of the task graph. The objective is to decrease the pipeline stage time interval as after filling up the pipeline, an instance execution of the application is performed each pipeline stage. The maximum number of stages is equal to the number of processors in the MPSoC system.

Our technique finds all the paths from the *dummy* start node to the *dummy* end node where the *dummy* start node is a node with an outgoing edge to all the nodes in the TDG with zero ingoing edges and the *dummy* end node is a node with an ingoing edge from all the nodes in the TDG with zero outgoing edges. Then it tries to remove some edges in the TDG to reduce the time on the critical paths. We find the critical paths based on the PEC values defined earlier. The removal of an edge means that the nodes at the subgraph corresponding to the head of the edge, SG_h, and that corresponding to the tail, SG_t, belongs to two separate stages in the pipeline. Any time an edge from T_i to T_j is removed, all the edges that connect SG_h and SG_t will be removed. The TDG can be at most divided into s unconnected graphs where s is the number of stages in the pipeline. Our task scheduling/memory partitioning heuristic will then be performed on the resultant TDG. An example of our pipeline technique is presented in Section 3.

3 Example

In this section, we present a task graph example to illustrate our heuristic as well as to show the effectiveness of integrating task scheduling and memory partitioning for embedded programs on a MPSoC. This task graph example is based on the *lame* benchmark from *MiBench* that consist of four tasks with their corresponding execution times in Mega cycles are shown in Figure 4(a) assuming no SPM. In this example, we assume a multiprocessor architecture of two homogeneous processors, 4 KB scratchpad memory, and unlimited off-chip memory.

Task scheduling and memory partitioning

1. Divide the application into tasks T_i.
2. Perform dependence analysis between tasks.
3. Construct the TDG based on dependence analysis and communication costs.
4. Divide the SPM memory equally between the processors.
5. **For** each task T_i and processor P_j, extract the following:
6. (i) Minimum computation time on P_j, Min_{ij}.
7. (ii) Maximum computation time on P_j, Max_{ij}.
8. (iii) Average computation time on P_j, Avg_{ij}.
9. Find ASAP for all the tasks based on Avg values.
10. L_1 = List of tasks in increasing order of ASAP.
11. **While** (L_1 not empty) **do**:
12. Get the first task T_f from L_1.
13. **For** each processor P_i:
14. Calculate the *elasticity* and PEC of P_i if T_f is mapped to P_i.
15. Find the minimum start time of T_f on P_i.
16. Find $END\_time(P_i)$ if T_f is mapped P_i.
17. **if** ($END\_time(P_i) < min$ || $PEC(P_i) <$ (1 - δ%)$PEC(P_j)$)
(Comment: P_j = processor corresponding to the current min value)
18. $min = END\_time(P_i)$
19. **else if** ($END\_time(P_i) = min$)
20. $min = END\_time$ of processor with the higher *elasticity*.
21. **End For**
22. Assign T_f to P_j corresponding to min.
23. Delete T_f from L_1.
24. Call Balance().
25. **End While**
26. **For** i = 1 to t **do**:
27. Call Balance().

Balance()

1. P_i = processor with maximum end time, $End\_time(P_i)$.
2. P_j = processor with minimum $End\_time(P_j)$*$elasticity$.
3. **while**($End\_time(P_j) < End\_time(P_i)$) **do**:
4. $Mem_i = Mem_i + \alpha\ Mem_j$.
5. $Mem_j = Mem_j$ - $\alpha\ Mem_j$.
6. Recompute().
7. **if** ($End\_time(P_j) \leq End\_time(P_i)$).
8. Perform the memory update.

Recompute()

1. Following the ASAP sort of scheduled tasks T_i and the new SPM budget distribution:
2. Recompute $time(T_i, Mem_j)$.
3. Recompute $Gain(T_i, Mem_j)$.
4. Recompute $Start\_time(T_i, P_j)$ where T_i is mapped to P_j of SPM = Mem_j.
5. Update the $Start\_time$ of all the tasks on P_j successor to T_i.
6. Recompute $End\_time(T_i, P_j)$.

Fig. 3. Our Task scheduling and memory partitioning heuristic

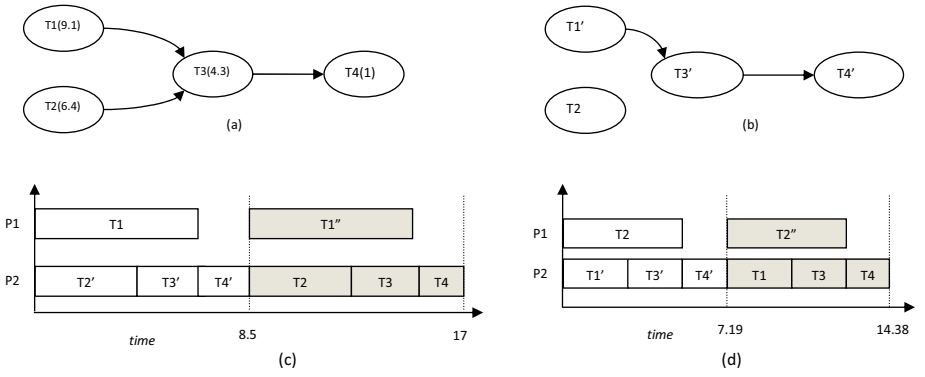


Fig. 4. (a) Original TDG. (b) TDG with pipelining. A solution using: (c) decoupled heuristics. (d) our heuristic.

Figure 4(a) shows the *lame* task graph with 4 tasks with data communications between tasks are represented by edges. We assume equal communication cost. Since task T_1 has the longest execution time with no SPM, usually current schedulers will map it into a separate processor. This is the solution that decoupled task schedule/memory partitioning heuristics will produce as they don't take into consideration the considerable reduction in computation time of T_1 with a bigger SPM memory. The solution is presented in Figure 4(c) with a total pipeline stage interval of 8.5. Task T_2 is of small *elasticity* which implies that adding more SPM memory to P_1 will not help much in reducing the execution time. Pipeline stage S_2 is represented by the tasks shown with dark fill. Tasks T_i, T_i', and $T"_i$ represents three instances of the same task from different runs of the application. In this solution 12.1 KB SPM memory is allocated to processor P_1 and the rest to P_2.

Since we have two processors in our MPSoC model, at most two pipeline stages are allowed. Our heuristic in Section 2 will find that there are two paths from the dummy source task to the dummy end task, p_{134} and p_{234}. Since there are only two processors, the parallel tasks T_1 and T_2 will be mapped to different stages in the pipeline. The important question now is whether to assign T_3 and T_4 to the same stage of T_1 or T_2. Based on the high *elasticity* value of T_1 compared to T_2 and based on the PEC values of p_{134} and p_{234}, our heuristic will map T_3 and T_4 to the same stage as T_1, namely S_2, since PEC of $p_{234} > PEC$ of p_{134}. The PEC of a path, say p_{234}, is calculated as $PEC(P_n)$ in Equation 2 assuming tasks T_2, T_3, and T_4 are mapped to processor P_n. After dividing tasks into different stages, our integrated task scheduling memory partitioning algorithm will be applied to the TDG in Figure 4(b).

Figure 4(d) shows our pipeline schedule with a pipeline stage of 7.19M cycles. This solution starts by assigning T_2 to P_1 and T_1' to P_2. After applying the *Balance*() procedure, 2.8 KB of SPM memory will be assigned to P_2 to balance the schedule as much as possible. The scheduler will then assign T_3' to P_2 since

its end time is lower on this processor as its *elasticity* is high. The *Balance*() subroutine will update the memory budget for each processor by moving 1 KB from P_1 SPM budget to P_2 to balance the schedule. T_4' will also be assigned to P_2 as its end time will be smaller on this processor at this step due to the high SPM memory budget allocated to P_2. After all the tasks are scheduled, the *Balance*() procedure will further reduce the cost by transferring the 0.2 KB SPM budget assigned to P_1 to P_2.

4 Experimental Results

We implemented five approaches to solve the task scheduling and memory allocation problem on MPSoC systems namely, (i) decoupled task scheduling and memory partitioning assuming equally partitioned SPM among all available processors $TSMP\_EQUAL$; (ii) decoupled task scheduling and memory partitioning with SPM partitioned among different processors with any ratio, $TSMP\_ANY$; (iii) our integrated task scheduling and memory partitioning heuristic described in Section 2, $TSMP\_INTEG$; (iv) our heuristic with pipelining $TSMP\_PIPE$; and (v) the optimal solution with pipelining based on the ILP formulation in [21], ILP_PIPE using the $CPLEX$ ILP solver [1]. We used several real life programs from the *Mediabench* and *MiBench*.

We used *Simplescalar* architectural simulation to profile the used benchmarks [3]. *Simplescalar* can simulate the execution of an application on a complex multiprocessor system on-chip architectures with different memory hierarchies. The MPSoC architecture used is similar to the one in Figure 1. The profiling is intended to (i) divide each application into computation blocks referred to as tasks, (ii) find the computation times for each task on each available processor in processor cycles, (iii) find the number of variables, (iv) the number of times each variable is used, $freq$, and (v) the size in bytes for each variable in the current application. The profiler information is based on a system with only off-chip memory. Using the profile information and dependence analysis, a task graph is constructed with a vertex for each task and an edge to represent the communication cost between two tasks. The communication cost depends on the size of data to be communicated between the two tasks and it is calculated through profiling. We assume a 100 cycle latency for off-chip memory access compared to 1 cycle latency for the SPM on-chip memory.

First we tested our techniques on the *enhance*, *lame*, *osdemo*, and *cjpeg* benchmarks. We tested those benchmarks assuming a multiprocessor system on chip of two processors and a scratch pad memory with size that varies between 4KB and 4 MB. We tested each of our benchmarks under three SPM budgets chosen based on the size of the benchmark. The choice of SPM sizes for each benchmark is essential as too little SPM or too much SPM for a certain embedded application may not reflect the effectiveness of our heuristic. The off-chip memory size is assumed to be unlimited that is it can hold all the data variables needed by the embedded application. We used an α of 10 meaning that 10% of SPM memory is being transferred between two tasks at a time in the *Balance*()

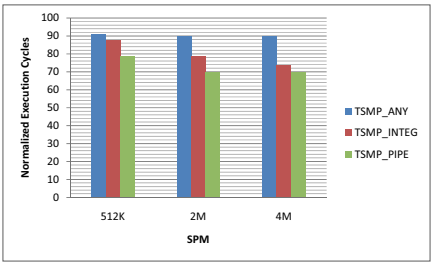


Fig. 5. Results for Pgp benchmark

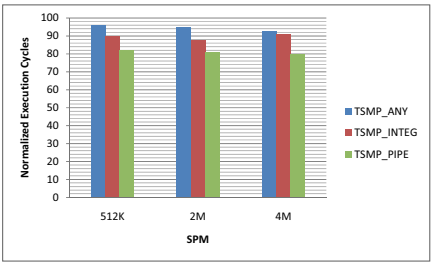


Fig. 6. Results for Rasta benchmark

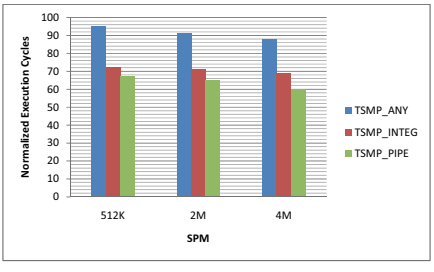


Fig. 7. Results for Pegwit benchmark

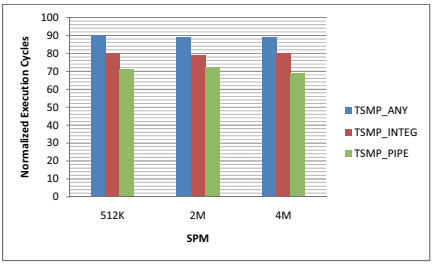


Fig. 8. Results for Epic benchmark

procedure. A smaller α may further improve the results with additional run time overhead.

The improvement greatly depends on the structure of the embedded application. $TSMP\_ANY$ improved over $TSMP\_EQ$ from little improvement close to 0% to dramatic improvement of 47%. Such improvements show that static memory allocation that is partitioning the SPM budget equally among the processors limits the effectiveness of SPM memories as it does not consider the characteristics of the tasks assigned to a processor in its memory partition decision.

Our integrated approach for task scheduling and SPM memory partitioning, $TSMP\_INTEG$, further improved the results over the decoupled approach, $TSMP\_ANY$. $TSMP\_INTEG$ improved over $TSMP\_ANY$ from little improvement close to 0% in some cases to dramatic improvement of 22%. This improvement is due to the guidance that our integrated approach uses to partition the memory based on the fact that the SPM configuration of a certain processor depends on the tasks mapped to that processor.

Our technique with pipelining, $TSMP\_PIPE$ is then tested. As expected, our embedded applications greatly benefit from pipelining as the execution time is decreased by 27% in some cases. The results emphasis the fact that such embedded applications can benefit significantly from pipelining. The pipeline cost is the computation time needed for one pipeline stage.

To show the effectiveness of our task scheduling/memory partitioning heuristic, $TSMP\_INTEG$, we compared it to an optimal integer linear formulation (ILP) based on the ILP formulation of this problem in [21], ILP_PIPE. The ILP solver is stopped after 35 minutes in some cases due to the long execution time taken by the ILP to produce optimal results. Following the same assumptions concerning the MPSoC system model and SPM memory budget, our $TSMP\_INTEG$ heuristic is in the range of 0% to 13% off the optimal solution in a negligible amount of time. This shows the effectiveness of our heuristic where in most of the cases our solution was close to the ILP one.

The ILP formulation is not scalable its running time is exponential in the number of variables in the application. For large scalar-based embedded application, the number of variables is usually large and thus the ILP will take very long time that makes the use of ILP infeasible for such applications. On the other side, our heuristic is of polynomial run time and thus it scales well with big applications. We tested our heuristic on the following four large embedded applications mainly, *pgp*, *rasta*, *pegwit*, and *epic*. Figures 5–8 show the results achieved by our heuristic when considering a system with 4 processors and an SPM budget ranging from 512K to 4M. The results in Figures 5–8 are the normalized execution cycles with respect to $TSMP\_EQ$. $TSMP\_ANY$, $TSMP\_INTEG$, and $TSMP\_PIPE$ improved over $TSMP\_EQ$ up to 12%, 33%, and 40% respectively. Keep in mind that a more aggressive data allocation techniques in SPM will further improve our results.

5 Related Work

Many research groups have studied the problem of task scheduling for applications on multiple processors with the objective is to minimize the execution time. Benini et al. [6] solved the scheduling problem using constraint programming and the memory partitioning problem using integer linear programming. The authors argued why those two choices fit the two problems the best. Kwok and Ahmed [12] presented a comparison among algorithms for scheduling task graphs onto a set of homogeneous processors on a diverse set of benchmarks to provide a fair evaluation of each heuristic based on a set of assumptions. De Micheli et al. [14] studied the mapping and scheduling problem onto a set of processing elements as a hardware/software codesign. Neimann and Marwedel [15] used integer programming to solve the hardware/software codesign partitioning problem. A tool for hardware-software partitioning and pipelined scheduling based on a branch and bound algorithm was presented in [7]. Their objective was to minimize the initiation time, the number of pipeline stages, and memory requirement. Cho et al. [8] proposed an accurate scheduling model of hardware/software communication architecture to improve timing accuracy.

Panda et al. [17,18] presented a comprehensive allocation technique for scratch-pad memories on uniprocessor to maximally utilize the available SPM memories to decrease the programs execution time. Optimal ILP formulations for memory allocation for scratch-pad memories were presented in [4,9]. An ILP formulation for

the SPM allocation problem to reduce the code size was presented in [19]. Steinke et al. [20] formulated the same problem with the objective to minimize the energy consumption. Angiolini et al. [2] optimally solved the problem of mapping memory locations to SPM locations using dynamic programming.

Many authors have studied the memory allocation problem in MPSoCs. The main focus of their research work is data parallelism in the context of homogeneous multiprocessor systems. Meftali et al. [13] formulated an ILP model of the memory allocation problem to obtain an optimal distributed shared memory architecture to minimize the global cost to access shared data as well as the memory cost. Kandemir et al. [11] presented a compiler-based strategy for optimizing energy and memory access latency of array dominated applications in a MPSoC. In [16], the authors proposed an ILP formulation for the memory partitioning problem on MPSoC. Suhendra et al. [21] studied the problem of integrating task scheduling and memory partitioning among a heterogeneous multiprocessor system on chip with scratch pad memory. This is the only paper, to the best of our knowledge, which addressed this problem in an integrated approach for MPSoC. They formulated this problem as an integer linear problem (ILP) with the inclusion of pipelining. Other works [5,10] have studied issues related to task scheduling and memory partitioning.

6 Conclusions

In this paper, we presented an effective heuristic that integrates task scheduling and memory partitioning on multiprocessor systems-on-chip with scratchpad memory. Compared to the widely-used decoupled approach, our integrated approach further improved the results since the appropriate partitioning of SPM spaces among different processors depends on the tasks scheduled on each of those processors and vice-versa. Results on several benchmarks from *Mediabench* and *MiBench* show the effectiveness of our approach compared to the decoupled approaches.

Acknowledgments. This work is supported in part by the U.S. National Science Foundation through awards 0121706, 0509442, 0541409 and 0811457.

References

1. Ilog inc., ilog cplex 8.1 reference manual, `http://www.ilog.com/products/cplex`
2. Angiolini, F., Benini, L., Caprara, A.: Polynomial-time algorithm for on-chip scratchpad memory partitioning. In: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems(CASES) (2003)
3. Austin, T., Larson, E., Ernst, D.: Simplescalar: An infrastructure for computer system modeling. IEEE Computer 35(2) (2002)
4. Avissar, O., Barua, R., Stewart, D.: An optimal memory allocation scheme for scratch-pad-based embedded systems. ACM Transactions on Embedded Computing Systems 1(1) (2002)

5. Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In: International Conference on Hardware-Software Codesign (CODES) (2002)
6. Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation and scheduling for mpsoc via decomposition and no-good generation. In: International Joint conferences on Artificial Intelligence (IJCAI) (2005)
7. Chatha, K.S., Vemuri, R.: Hardware-software partitioning and piplined scheduling of transformative applications. IEEE Transactions on VLSI 10(3) (2002)
8. Cho, Y., Zergainoh, N.-E., Yoo, S., Jerraya, A., Choi, K.: Scheduling with accurate communication delay model and scheduler implementation for multiprocessor system-on-chip. Design Automation for Embedded Systems (2007)
9. Dominguez, A., Udayakumaran, S., Barua, R.: Heap data allocation to scratch-pad memory in embedded systems. Journal of Embedded Computing (2005)
10. Kandemir, M., Dutt, N.: Memory systems and compiler support for mpsoc architectures. Multiprocessor Systems-on-Chips (2005)
11. Kandemir, M., Ramanujam, J., Choudhury, A.: Exploiting shared scratch pad memory space in embedded multiprocessor systems. In: Design Automation Conference (DAC) (2002)
12. Kwok, Y.-K., Ahmad, I.: Benchmarking and comparison of the task graph scheduling algorithms. Journal of Parallel and Distributed Computing 59(3) (1999)
13. Meftali, S., Gharsalli, F., Rousseau, F., Jerraya, A.: An optimal memory allocation for application-specific multiprocessor system-on-chip. In: International Symposium on Systems Synthesis (ISSS) (2001)
14. De Micheli, G., Ernst, R., Wolf, W.: Readings in hardware/software co-design. Morgan Kaufmann, San Francisco (2002)
15. Neimann, R., Marwedel, P.: Hardware/software partitioning using integer programming. In: Design Automation and Test in Europe (DATE) (1996)
16. Ozturk, O., Kandemir, M.: An integer linear programming based approach to simultaneous memory space partitioning and data allocation for chip multiprocessors. In: IEEE computer society Annual Symposium on VLSI (ISVLSI) (2006)
17. Panda, P., Dutt, N., Nicolau, A.: Memory issues in embedded systems-on-chip: optimization and exploration. Kluwer Academics Publisher, Dordrecht (1999)
18. Panda, P., Dutt, N.D., Nicolau, A.: On chip vs off chip memory: the data partitioning problem in embedded processor-based systems. ACM Transactions on Design Automation of Electronic Systems (TODAES) 5(3) (2000)
19. Sjodin, J., Von Platen, C.: Storage allocation for embedded processors. In: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) (2001)
20. Steinke, S., Wehmeyer, L., Lee, B.-S., Marwedel, P.: Assigning program and data objects to scratchpad for energy reduction. In: Design Automation and Test in Europe (DATE) (2002)
21. Suhendra, V., Raghavan, C., Mitra, T.: Integrated scratchpad memory optimization and task scheduling for mpsoc architecture. In: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) (2006)

Hybrid Super/Subthreshold Design of a Low Power Scalable-Throughput FFT Architecture

Michael B. Henry and Leyla Nazhandali

Virginia Polytechnic Institute and State University,
302 Whittemore , Blacksburg, VA 24061
{mbh,leyla}@vt.edu

Abstract. In this paper, we present a parallel implementation of a 1024 point Fast Fourier Transform (FFT) operating at subthreshold voltage, which is below the voltage that turns the transistors on and off. Even though the transistors are not actually switching as usual in this region, they are able to complete the computation by modulating the leakage current that passes through them, resulting in a 20-100x decrease in power consumption. Our hybrid FFT design partitions a sequential butterfly FFT architecture into two regions, namely memory banks and processing elements, such that the former runs at superthreshold and the latter in the subthreshold voltage region. For a given throughput, the number of parallel processing units and their supply voltage is determined such that the overall power consumption of the design is minimized. For a 1024 point FFT operation, our parallel design is able to deliver the same throughput as a serial design, while consuming 65% less power. We study the effectiveness of this method for a variable throughput application such as a sensor node switching between a low throughput and high throughput mode, e.g. when sensing an interesting event. We compare our method with other methods used for throughput scaling such as voltage scaling and clock scaling and find that our scaling method will last up to three times longer on battery power.

1 Introduction

As Charles Van Loan wrote in his book "The Fast Fourier Transform (FFT) is one of the truly great computational developments of this century. It has changed the face of science and engineering so much that it is not an exaggeration to say that life as we know it would be very difficult without the FFT" [13]. The FFT has had a widespread application in traditional fields such as communication and manufacturing. The advent of wireless sensor networks has created even more applications for the transform. Sensor nodes are employed to monitor the environment and report interesting data or significant events[1]. The FFT can be used to analyze the raw data in order to identify such events. This is especially important for situations where large amounts of data are collected, but there is only an occasional need to report back data. Since the sensor nodes run on either a battery or a limited amount of scavenged energy, and communication costs are

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 278–292, 2009.

still the dominant factor in power consumption, it is usually advantageous to process the data locally and transmit only a message if an interesting event is detected. In an example scenario, a node with an acoustic sensor, which is employed in a field to detect passing vehicles, collects sound samples periodically. It then analyzes the collected data using an FFT to determine if its frequency content includes components representing frequencies found in a moving vehicle such as a humming engine frequency. If so, it records a significant event and transmits the data to a central station or other nodes depending on how the sensor network is implemented.

It is also highly desirable to be able to scale the throughput of an FFT operation for a sensor node[8]. Increasing the amount of transformed data yields more resolution in the frequency domain. However, high quality FFTs are computationally intensive and consume high levels of power, which is impractical for wireless sensor nodes. It is therefore suggested that during idle periods, a low throughput FFT is used that consumes less power. When a significant event is suspected, the throughput of the FFT is ramped up so that the data can be analyzed more diligently.

In this paper, we present a low-power parallel implementation of the FFT with scalable throughput. Our novel FFT design partitions a traditional butterfly FFT architecture into two regions, namely memory banks and processing elements, so that the operating voltage of the former region is above the threshold voltage while that of the latter region is below the threshold voltage, which is the voltage that turns the transistors on and off. Above this threshold, transistors operate similar to a switch that let the current flow to either charge or discharge the load. Below the threshold voltage the transistors are not actually switching as usual; instead they are able to complete the computation by modulating the leakage current that passes through them, resulting in a 20-100x decrease in power consumption.

Our proposed design is able to deliver the same throughput as a traditional design while consuming 65% less power. Furthermore, we study the effectiveness of this method in a variable throughput application. We compare our method with other methods used for scaling the throughput, namely voltage scaling and clock scaling. Our results indicate that in an example scenario, if all these designs are running on 2 alkaline AA batteries and spend 15% of their time in high quality mode and the rest in the low quality mode, our design can last up to 111 days while the other two last only 59 and 40 days respectively. The cost of the decrease in power is a 5x increase in area.

The rest of this paper is organized as follows: Section 2 describes the background related to this paper, which includes two subsections: one on the fundamentals of FFT operation and the other on the basics of subthreshold operation. In Section 3, we present our novel parallel FFT architecture based on the traditional butterfly design. Section 4 describes our employed methodology to carry out the experiments while Section 5 presents the results of these experiments. Section 6 presents related work. Finally, in Section 7 we present the conclusions and future directions of this study.

2 Background

2.1 Fast Fourier Transform

The Fast Fourier Transform (FFT), formulated by Cooley and Tukey[6], is an efficient method for calculating the frequency content of a signal. The number of samples in the signal, N, determines the frequency resolution and quality of the Fast Fourier Transform. An increase in the number of data points yields more frequency resolution, but takes more computation as the complexity of this transform is equal to $O(N * log_2(N))$. Equation 1 presents the formulas that define the FFT.

$$X_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} W^{(2m)k} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} W^{(2m+1)k} \tag{1}$$

$$W^n = e^{-\frac{2\pi i}{N} n} \tag{2}$$

Equation 1 breaks up an N point FFT into the sum of two $\frac{N}{2}$ point FFTs. These $\frac{N}{2}$ point FFTs can then be broken up again and again leading to a fast recursive implementation of a DFT. The W coefficients are constants equal to nth roots of unity in the complex plane, traditionally called twiddle factors. Figure 1a shows the signal flow graph of an 8-point FFT. There are 3 levels in an 8-point FFT, corresponding to the $log_2(N)$ term in the complexity. There is a repetitive pattern in the FFT signal flow graph that looks like a butterfly. Each butterfly contains a complex addition, subtraction, and multiplication. It can be seen from the figure that there are 4 butterfly computations in each level, which is half of 8 data points and corresponds to the N term in the complexity. Overall there are $(N * log_2(N))/2$ butterfly operations in this FFT implementation[1]. It

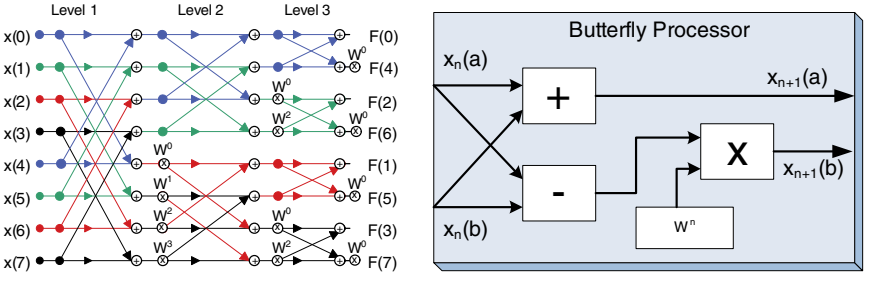


Fig. 1. (A): Signal Flow Graph of 8 pt. FFT (B): Butterfly processor

[1] There are other ways to implement FFT. However, the Cooley-Tukey algorithm discussed in this paper is by far the most common implementation of FFT due to its efficiency.

is highly impractical to implement this signal flow graph in hardware for a 1024-point FFT due to its large area, so hardware must be reused to perform these calculations. One possibility is to use a single butterfly element and compute all the $(N * log_2(N))/2$ butterfly operations serially. Intermediate values can be stored and recalled in a memory bank.

2.2 Subthreshold Voltage Operation

Figure 2(a) shows a CMOS transistor identifying its source and gate. When the source voltage is above a certain ***threshold***, the transistor effectively functions like a switch responding to the changes that come from gate voltage. Lowering the voltage source or voltage scaling has been a prevalent method for improving the energy efficiency of microprocessors [5,2]. This is due to the fact that reducing the voltage drops the energy consumption of a microprocessor quadratically, while decreasing its performance linearly. The lower limit for voltage-scaling has typically been restricted to half the nominal voltage - the voltage that hardware is designed to typically operate at. Until recently, this limit has only been imposed upon by a few sensitive circuits with analog-like operation such as sense amplifiers. However, it has been known for some time that standard CMOS gates operate seamlessly from full-voltage source to well below the threshold voltage - the voltage that turns the transistor on - at times reaching as low as 100mV [10,14]. Recently, a number of prototype designs have demonstrated that with careful design and replacement of these analog-like devices with standard switching counterparts, it is possible to extend the traditional voltage-scaling limit to below the threshold voltage, i.e., ***subthreshold-voltage***[2] region [19,18,11].

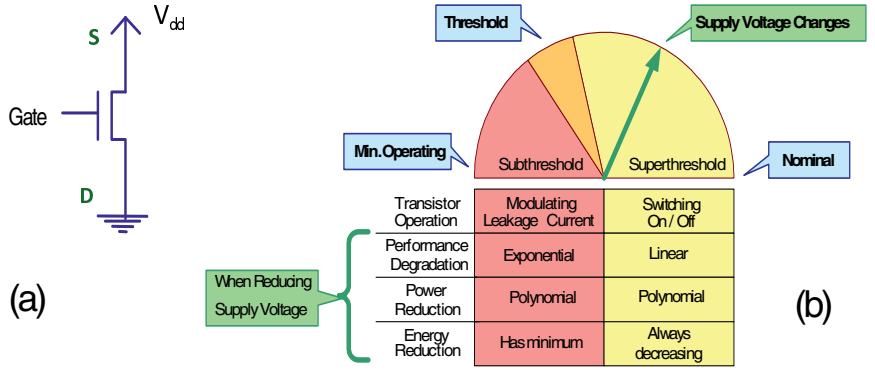


Fig. 2. Overview of sub/superthreshold operation

Figure 2(b) provides an overview of subthreshold and superthreshold operation differences. First, the transistors are not switching as normal in the subthreshold region; instead they modulate the use the changes in the leakage current that

[2] In this document, we may use super/subthreshold words in place of super/subthreshold-voltage for brevity.

passes through to charge or discharge their load and eventually perform computation. This, in turn, results in exponential degradation of performance in the subthreshold region as opposed to linear degradation when voltage is reduced in superthreshold[3]. Moreover, because the system operates at much lower voltages, it becomes more susceptible to some manufacturing and operational problems such as process variation and soft errors. These issues as well as accurate modeling of subthreshold leakage are currently under investigation by several research groups in the VLSI and digital electronics area who have shown promising results [17,12,9,19]. The focus of this paper is not providing a solution to some of these known problems for subthreshold operation. Instead, we present an example of the new opportunities this technology provides to designers by showcasing an architecture for FFT with scalable throughput and ultra low power consumption.

3 Implementation

The 1024 point FFT architecture used in this paper is based on [16] and is implemented in a 90nm technology. It consists of three major modules: RAM, processing element, and ROM. The processing element (PE) performs the FFT calculations and contains at least one butterfly processor shown in Figure 1. The butterfly processor implements a complex addition, subtraction, and multiplication. The multiplication is implemented using a booth multiplier and the addition is implemented using efficient carry select adders. The architecture processes 32 bit complex numbers, where 16 bits represent the real portion and 16 bits represent the complex portion, all in the Q15 fixed point format.

The ROM module stores the constant twiddle coefficients while the RAM module is responsible for storing inputs, outputs and intermediate values. The PE takes in three inputs: the first two inputs are either the input signals, or the intermediate values that are result of a previous butterfly operation. The third operand is a constant W^n coefficient, taken from the ROM. The W^n coefficients are arranged around the unit circle in the complex plane, but the FFT only uses coefficients with positive imaginary parts. There is also symmetry of the coefficients on the left half and right half of the unit circle, so overall 256 32-bit coefficients are needed in the ROM for a 1024 point transform. The architecture implements the FFT using decimation in frequency, which means initially, the RAM holds the 1024 32-bit input values, and they get replaced with intermediate and eventually output values as the FFT progresses.

3.1 Parallelization and Throughput Scalability

It is well-known that the parallelism present in applications such as FFT can be utilized to improve the energy efficiency of the system without sacrificing performance [4]. In our design, we exploit the unique characteristics of subthreshold

[3] In order to obtain power and frequency trends for the butterfly processor in the subthreshold region, a post layout simulation was performed using extracted parasitics and a Fast-SPICE program.

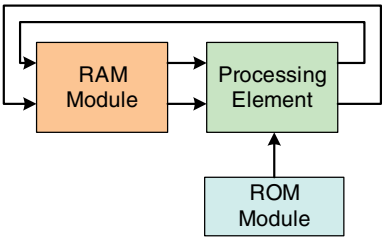


Fig. 3. FFT Architecture

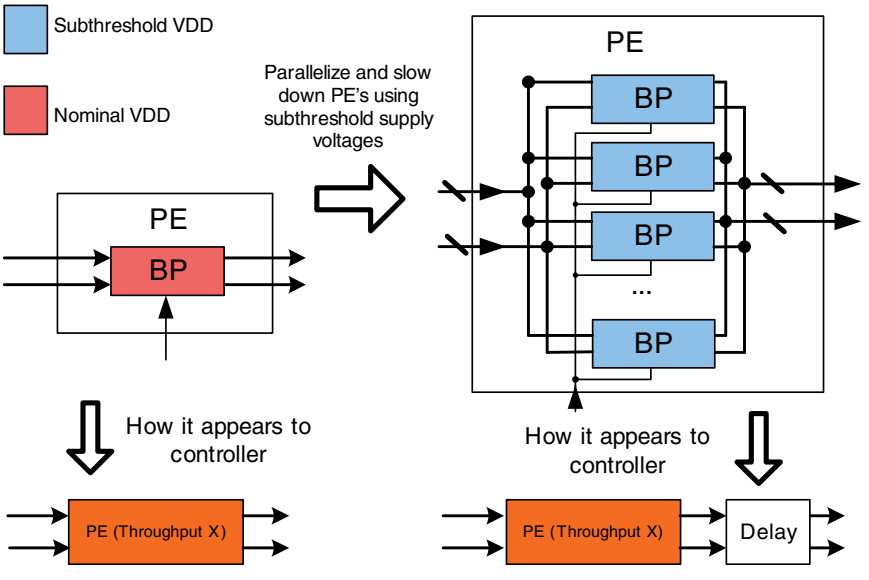


Fig. 4. Parallelization of the PEs: Butterfly processors are added to the processing element and their supply voltage is reduced so the throughput remains the same. To the controller, this only appears as a delay from the input to the output.

operation in this context. Traditional parallel designs focus on increasing the throughput of a system. Our design exploits parallelism in a different way, by adding parallel units and slowing them down to match the original throughput. To slow down the parallel units, the supply voltage is reduced which greatly reduces the power consumption. We will show that the optimal supply voltage for the parallel FFT architecture is in the subthreshold region.

Our goal is to design an FFT architecture to operate with minimum energy given a desired level of performance. In order to achieve this, the number of butterfly processors (BPs) within the processing element (PE) is increased and the supply voltage to the processors is decreased such that the throughput remains constant, as shown in Figure 4. Even though a processing element may contain a number of butterfly processors, it must still only have three inputs and

two outputs. The processing element, therefore, needs a data distribution bus that distributes the serial data from the RAM to each butterfly processor, and a collection bus that forms the processed data back into a serial stream. The PE uses a demultiplexer and a multiplexer to accomplish the data distribution and collection respectively. Data is distributed in a staggered fashion to the BPs, and is collected on the other end in a staggered fashion. The supply voltage of the BPs is scaled such that the existing data in a BP is processed before new data is sent to it. The PE, therefore, exhibits the same throughput no matter how many BPs it contains. The only difference is that there is increased latency from the input of the a piece of data to the collection on the other side, which grows linearly with respect to the number of BPs.

The parallelization lends itself well to a scalable throughput design. At maximum throughput, all of the BPs in the PE are activated and are processing data. To reduce the throughput, only some of the BPs are active and the rest are powered off. Ideally, the operating voltage of these remaining BP's should be adjusted to achieve maximum energy efficiency. However, as we will show in section 5, the benefits of this ideal method does not justify the extra burden of adding voltage scaling capability to the circuitry. Instead, we simply power off a certain number of BP's while leaving the rest to run at the same speed. We call this method *active unit scaling*.

With more than one BP running in parallel, one can envision a design where the RAM bank is split into multiple banks, each running at a lower speed than the original RAM and feeding the slow-running BP units. As an example, the 1024-word RAM bank can be split into two separate 512-word banks. Both of these RAM modules, i.e. the single bank or dual bank, are capable of supplying data at the same rate, but each 512-word RAM banks is now required to run at half the speed of the original one. Because of the relaxed speed requirement, the supply voltage of the two 512-word banks can be reduced, resulting in reduction of the RAM power. The paid penalty is the area taken up by the additional controller logic in the second bank. We study the practicality of this method and present our findings in the results section.

4 Methodology

Table 1 shows the tool chain used for synthesis and simulation of our design. In the rest of this section, we present the detailed methodology for the two major experiments done to study our proposed design.

4.1 Minimum Energy Architecture Experiment

The purpose of this experiment is to compute the optimum number of parallel butterfly processors as well as their operating voltage in order to achieve minimum energy consumption for a given throughput. Since the throughput is kept constant, the energy consumption is directly related to the power and we use these two terms interchangeably. To carry out this experiment, we use the trends

Table 1. Tool Chain

HDL Language	Verilog
HDL Simulator	Synopsys VCS
Technology	UMC Standard Performance 90nm
Standard Cell Library	Faraday UMC 90nm Generic Core 1.0V, RVT
RAM/ROM memory compiler	Faraday 90nm Memmaker
Voltage Scaling Characterization	Synopsys HSPICE, NanoSim
Power Simulation	Synopsys Primetime PX

presented in Section 2.2, which give the power consumption and the frequency (or throughput) of a BP for a given supply voltage. The inverse of this is used to determine the supply voltage - and then, power consumption - of a BP running at a given throughput. The throughput of the processing element with a single butterfly processor at nominal voltage is referred to as $TP_{Nominal}$, and is the target throughput for the minimum energy architecture. It is possible to increase the number of BPs in the PE and maintain $TP_{Nominal}$ by reducing the supply voltage of the BPs. Herein, n will refer to the number of parallel BPs in the PE. In order to achieve $TP_{Nominal}$ in a processing element, the throughput of each individual butterfly processor must be $\frac{TP_{Nominal}}{n}$. Using the characterization curves, the supply voltage that yields a BP throughput of $\frac{TP_{Nominal}}{n}$ will be determined and the power consumption at that speed will be noted. The total power consumption of the PE is the power consumption of a BP at $\frac{TP_{Nominal}}{n}$ throughput times n. The power consumption of the bus needed to distribute and collect data from n BPs will also be determined and added to the BPs' power consumption. The *minimum energy architecture* will be defined as the FFT architecture with n BPs that consumes the least amount of power, while maintaining a throughput of $TP_{Nominal}$.

4.2 Throughput Scaling Experiment

As stated in the introduction, there is a desire to dynamically scale the throughput of the FFT architecture. During idle times, less data points can be used in the FFT, and the throughput of the FFT processor can be reduced. Figure 5 shows an example scenario in a wireless sensor network. From $t0$ to $t1$, the hardware is performing a 256 point FFT. The throughput of the hardware is one quarter of the throughput at max speed. At $t1$, an event is detected and the hardware shifts to a 1024 point FFT. At this point, the hardware is running at its maximum throughput. From $t2$ to $t3$, the hardware goes back to a 256 point FFT. The duty cycle of the FFT hardware will be defined by Equation 3, or the time the hardware is spent at max throughput divided by total time.

$$DutyCycle = \frac{t_2 - t_1}{t_3 - t_0} \tag{3}$$

There are three different methods that can be used to reduce the throughput of the FFT hardware and save power. The first method is clock scaling, where the

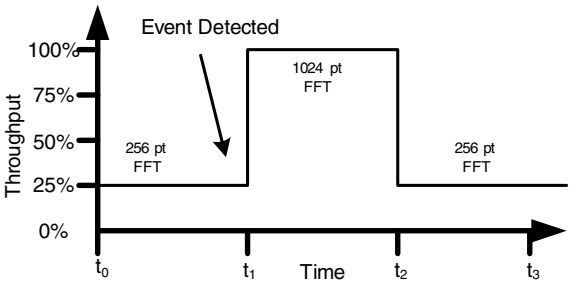


Fig. 5. Example sensing scenario

clock speed to the PE is reduced. The supply voltage remains at nominal 1V. This will be done to the $n = 1$ FFT architecture (one BP per PE). The second method is dynamic voltage scaling, where the clock *and* supply voltage to the PE is reduced. This will also be done to the $n = 1$ FFT architecture. The third method will be the active unit scaling mentioned in Section 3.1. The minimum energy architecture will be used for this method. The ROM and RAM are kept running at nominal voltage for this experiment. The three methods of voltage scaling will be tested over various throughputs and the power consumption will be determined. It will also be determined, for a given duty cycle defined by Eq. 3, how long each of the throughput scaling methods can last on two alkaline AA batteries (at 1500mAH each) while executing the sensing scenario in Figure 5.

5 Results

5.1 Synthesis Results

Table 2 shows the results of the memory compiler and the synthesized verilog code at nominal voltage (1V) and maximum speed.

Table 2. Synthesis Results of FFT Architecture

Butterfly critical path	4.43ns
Butterfly throughput	222MHz
Butterfly power consumption	19.6mW @ 222MHz
Butterfly area	0.0498mm^2
RAM access time	0.89ns
RAM power	6.22mW @ 444MHz
RAM area	0.126mm^2
ROM access time	0.74ns
ROM power	1.13mW @ 111MHz
ROM area	0.024mm^2

5.2 Minimum Energy Architecture Experiment Results

Figure 6b shows the power consumption (not including bus overhead) of the processing element(PE) for a given number of butterfly processors (BP) inside the PE. The throughput is kept constant at 222MHz while decreasing the supply voltage. The power decreases dramatically at first, but as the supply voltage reaches the subthreshold region, there are diminishing returns for adding more elements. As the number of BPs increase, the power consumption of the bus increases due to the increased load capacitance and increased number of flip flops required for distribution and collection of data. Figure 7a shows the data distribution and collection overhead for a given number of butterfly processors. The power consumption of the PE decreases with additional BPs while the power consumption of the bus increases with additional BPs, so there is a point where the PE reaches its maximum power efficiency for the targeted throughput. Since the throughput is kept constant, all the power savings translate to energy savings. Figure 7b shows the total power consumption of the entire FFT architecture. This includes the RAM, ROM, and controller power. According to the figure, the minimum energy is achieved at $n = 32$.

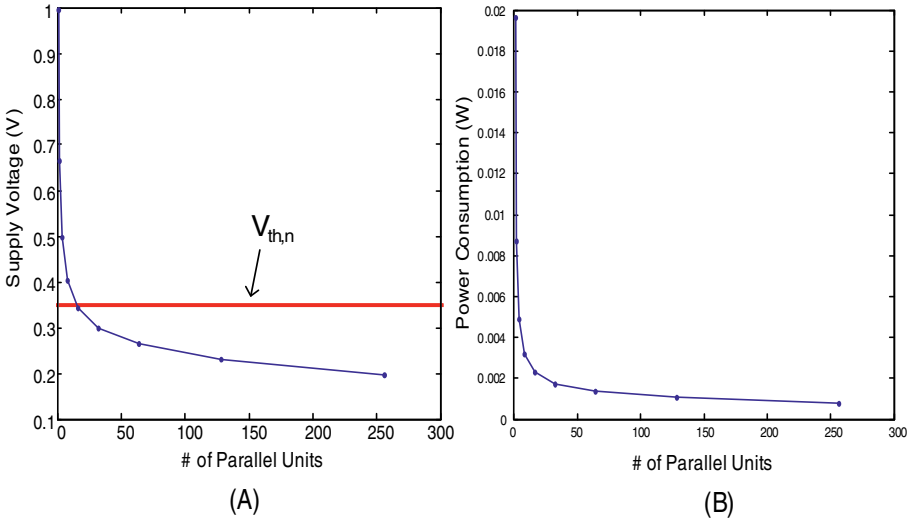


Fig. 6. (A): Supply voltage for a given number of BPs. (B): Power consumption of the PE for a given number of BPs, not including bus overhead for a constant throughput of 222MHz.

5.3 Throughput Scaling Experiment Results

Figure 8a shows the results of the throughput scaling in terms of power consumption. The clock scaling and the supply voltage scaling methods are identical at 100% duty cycle operating at nominal voltage of 1V and running at maximum speed of 222 MHz. However, at lower duty cycles, the supply voltage scaling does

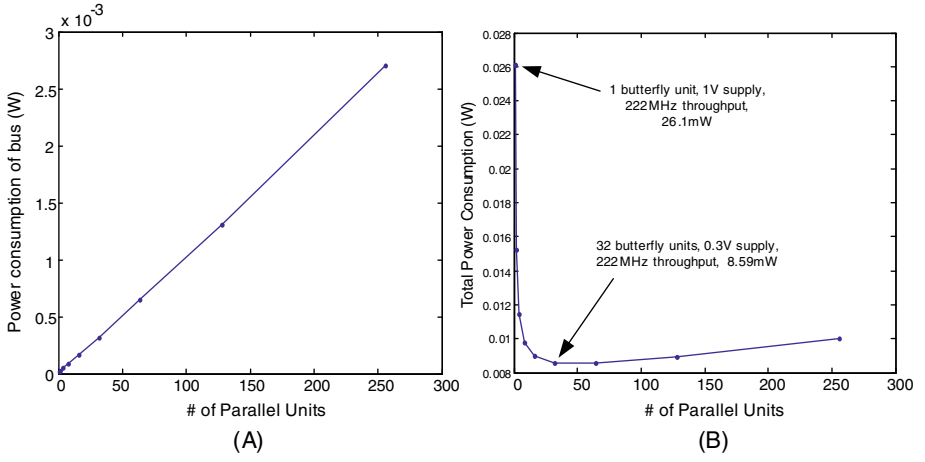


Fig. 7. (A): Power consumption of the data distribution and collection bus within the PE. (B): Total power consumption of the entire architecture for a given number of BPs including RAM, ROM, bus, and controller.

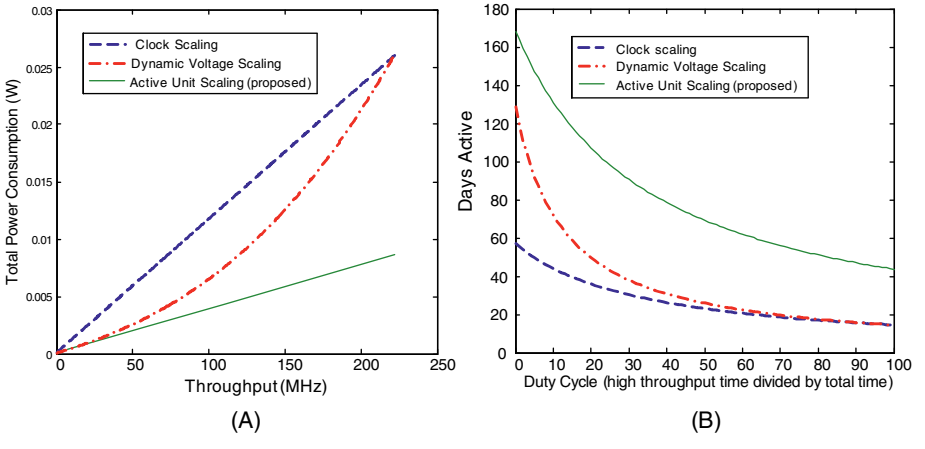


Fig. 8. (A) Total power consumption of our proposed scalable-throughput design compared to a single-butterfly design, whose throughput is scaled using two traditional methods, namely voltage scaling and clock scaling with respect to throughput (B) Number of days the above designs can remain active on 2 AA batteries with respect to duty cycle

better than clock scaling as it reduces the power quadratically. Meanwhile, our proposed active unit scaling method outperforms both traditional methods in all cases. Figure 8b shows the results of the experiment mentioned in 4.2, where the throughput is varied between a 256 point FFT and a 1024 point FFT at various duty cycles. Figure 8b presents the number of days our design, as well as two other traditional designs, can survive while running off two heavy duty

Table 3. The comparison between ideal throughput scaling and our active unit scaling to achieve 28Mhz throughput

	Active BPs	Supply (V)	Processing Power (mW)
Ideal Throughput Scaling	8	0.264 V	1.153mW
Active Unit Scaling	4	0.3V	1.168mW

Alkaline AA batteries. The proposed active unit scaling architecture lasts longer than the other two architectures for all possible duty cycle scenarios. As shown in the Figure, at around 30% duty cycle, our proposed architecture using active unit scaling can last up to 3 months while the other two methods can barely survive beyond a month.

As mentioned in Section 3.1, the ideal method of throughput scaling is to adjust both the supply voltage and the number of active BPs. In other words, to achieve maximum energy efficiency while scaling the throughput, one has to identify the optimum number of BPs and their operating voltage for the desired throughput as shown in Section 5.2 for 222MHz 1024 point FFT. Instead, we propose a simple method to simply turn off a pre-calculated number of BP's to achieve the same result. Table 3 shows the proposed active unit scaling vs. the ideal method of throughput scaling at 28MHz, or one eighth of the maximum throughput. It is shown that the proposed method of throughput scaling only consumes 1.3% more power than the ideal method. In addition, the ideal throughput scaling method involves more control logic and fine tune control of the supply voltage. The proposed scaling method, on the other hand, only requires that BPs be shut down without changing the supply voltage.

5.4 Comparisons

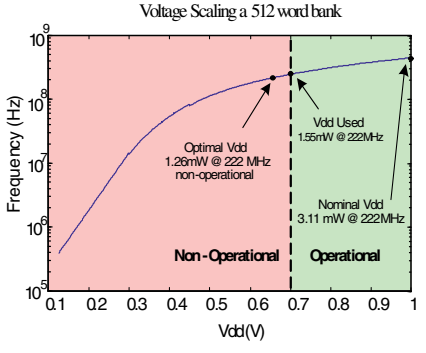
Table 4 compares the non parallelized FFT architectures with two parallelized FFT architectures. While $n = 32$ constitutes the minimum energy architecture, the $n = 16$ design provides a sweet spot with a slight increase in power and half of the area overhead. The total power includes processing, RAM, ROM, and the bus.

Table 4. Comparison of different number of BPs

	$n = 1$	$n = 16$	$n = 32$
Total Power @ 222MHz	26.11mW	9.00mW	8.54mW
Total Area	0.200mm^2	0.947mm^2	1.74mm^2
PE Supply Voltage	1.0V	0.344V	0.30V
Days on 2 AA bat. @ Max Duty Cyc.	14	42	44
Days on 2 AA bat. @ 15% Duty Cyc.	40(CS), 59(DVS)	111	118

5.5 Splitting the RAM Bank

Figure 9(a) shows how the frequency of a 512-word 32-bit RAM bank scales with the supply voltage, which results in significant power savings. However, in practice, it is known that regular 6T SRAM cells are not able to function below 0.7V in 90nm technology [20][4]. Therefore, even though splitting the RAM bank ideally calls for lowering the voltage, we cannot go beyond 0.7V. The table in Figure 9 shows the result of our analysis. It can be seen that given the restriction on lowering the supply voltage, splitting the RAM into two banks provides the most power efficient design by reducing the memory power consumption by more than half while incurring little area overhead.



# of Banks	Word Size per bank	Ideal Supply Voltage	Actual Supply Voltage	Total RAM Power	Total RAM Area
1	1024	1.0V	**1.0V**	6.22 W	. 124 mm^2
2	512	0.666V	**0.7V**	3.10 W	. 156 mm^2
4	256	0.499V	0.7 V	4.60 W	. 196 mm^2

Fig. 9. Splitting the RAM bank in half to 512 words and reducing the supply voltage

6 Related Work

[19] presented a 180mv FFT architecture that runs in the subthreshold region. The entire architecture runs at subthreshold voltage and the resulting throughput is very low due to a 164 Hz clock cycle at 180mV and 10kHz at 350mV. Our design, on the other hand, only runs the processing elements at subthreshold voltage, and parallelizes the processing to achieve a throughput typical of nominal voltage FFT processors. In addition, the architecture in [19] would have to use dynamic voltage scaling to achieve a higher throughput, and it was shown that our active unit scaling architecture outperforms dynamic voltage scaling. [15] presents a general purpose processor for the purposes of wireless sensor networks that runs in the subthreshold region. The processor is only capable of executing simple commands and a program to execute an FFT operation would be very slow due to the low clock frequency and the lack of FFT specific hardware. [7] presented a design methodology for parallel cores running at near threshold voltage. The cores were general purpose processors with caches, and they executed parallel software routines. Our work is different in that we present minimum energy hardware for a specific application.

[4] Currently, there are designs of SRAM cells that are functional near or below threshold voltage. However, they require additional transistors resulting in some additional power consumption [3].

7 Conclusion

In this paper, we presented a novel FFT architecture based on the traditional butterfly-based FFT architecture, and greatly reduced its power consumption by exploiting parallelism. Additional butterfly processors were added, and their supply voltage was reduced while keeping the overall throughput constant. We showed that the optimum operating voltage of butterfly processors is well below the threshold voltage at around 0.3V. The transistors still operate at this voltage, not by turning on and off, but by modulating the leakage current passing through them. The RAM module of the design, however, was kept to operate in superthreshold at 0.7V maximizing power efficiency while keeping the SRAM cells functional, resulting in a hybrid super/subthreshold design. By exploiting the parallelism in the processing element of the architecture, we were able to achieve 68% reduction in total power consumption of the FFT architecture. A 10% reduction is obtained by splitting the SRAM bank into two banks and scaling their voltage from nominal 1V to 0.7V. This total reduction of 78% is obtained at the expense of increasing the area of the design by about 5 times. Even though the area increase may seem significant, the reduced cost of silicon and the increased need for ultra low power applications such as wireless sensor networks promotes such design directions.

In addition, we proposed an efficient method to enable scaling the throughput of our design by turning off some of the active subthreshold parallel units without changing their supply voltage. We compared our method to traditional methods of clock scaling and dynamic voltage scaling by simulating the designs running on two AA batteries. Over the entire range of possible duty cycles of high throughput to low throughput, the active unit scaling outlasted the other two methods by a wide margin. At 15% duty cycle, the active unit scaled design lasted 111 days, compared to 59 and 40 days for dynamic voltage scaling and clock scaling respectively. Future work would include taking advantage of the university program of UMC foundry by fabricating a complete ASIC design of our proposed FFT architecture. This will provide us with the realistic restrictions imposed on our design, which will help us to make refinements and enhancement to it. Moreover, we intend to develop a general model for estimating how various architectures will benefit from the minimum energy architecture and active unit scaling methods detailed in this paper.

References

1. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Computer Networks 38(4), 393–422 (2002)
2. Burd, T.D., Brodersen, R.W.: Energy efficient cmos microprocessor design. In: HICSS 1995: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS 1995), Washington, DC, USA, p. 288. IEEE Computer Society, Los Alamitos (1995)
3. Calhoun, B.H., Chandrakasan, A.: A 256kb sub-threshold SRAM in 65nm CMOS. In: IEEE International Solid-State Circuits Conference, 2006, ISSCC 2006. Digest of Technical Papers, pp. 2592–2601 (February 2006)

4. Chandrakasan, A.P., Brodersen, R.W.: Low Power Digital CMOS Design. Kluwer Academic Publishers, Norwell (1995)
5. Chandrakasan, A.P., Sheng, S., Brodersen, R.W.: Low-power CMOS digital design. IEEE Journal of Solid-State Circuits 27(4), 473–484 (1992)
6. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Mathematics of Computation 19(90), 297–301 (1965)
7. Dreslinski, R.G., Zhai, B., Mudge, T., Blaauw, D., Sylvester, D.: An energy efficient parallel architecture using near threshold operation. In: 16th International Conference on Parallel Architecture and Compilation Techniques, PACT 2007, pp. 175–188 (2007)
8. Heinzelman, W.R., Sinha, A., Wang, A., Chandrakasan, A.P.: Energy-scalable algorithms and protocols for wireless microsensornetworks. In: IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2000. Proceedings, Istanbul, Turkey, vol. 6, pp. 3722–3725 (2000)
9. Jayakumar, N., Khatri, S.P.: A variation tolerant subthreshold design approach. In: DAC 2005: Proceedings of the 42nd annual conference on Design automation, pp. 716–719. ACM Press, New York (2005)
10. Kao, J., Narendra, S., Chandrakasan, A.: Subthreshold leakage modeling and reduction techniques. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517. Springer, Heidelberg (2002)
11. Kim, C.H.I., Soeleman, H., Roy, K.: Ultra-low-power DLMS adaptive filter for hearing aid applications. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 11(6), 1058–1067 (2003)
12. Kim, T.-H., Eom, H., Keane, J., Kim, C.: Utilizing reverse short channel effect for optimal subthreshold circuit design. In: ISLPED 2006: Proceedings of the 2006 international symposium on Low power electronics and design, pp. 127–130. ACM Press, New York (2006)
13. Loan, C.V.: Computational frameworks for the fast Fourier transform. Society for Industrial and Applied Mathematics, Philadelphia (1992)
14. Meindl, J.D., Davis, J.A.: The fundamental limit on binary switching energy for terascale integration (TSI). In: IEEE JSSCC, vol. 35 (February 2002)
15. Nazhandali, L., Zhai, B., Olson, J., Reeves, A., Minuth, M., Helfand, R., Pant, S., Austin, T., Blaauw, D.: Energy optimization of subthreshold-voltage sensor network processors. SIGARCH Comput. Archit. News 33(2), 197–207 (2005)
16. Pirsch, P.: Architectures for Digital Signal Processing. Wiley, West Sussex (1998)
17. Raychowdhury, A., Paul, B., Bhunia, S., Roy, K.: Computing with subthreshold leakage: device/circuit/architecture co-design for ultralow-power subthreshold operation. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 13(11), 1213–1224 (2005)
18. Sze, V., Blazquez, R., Bhardwaj, M., Chandrakasan, A.: An energy efficient subthreshold baseband processor architecture for pulsed ultra-wideband communications. In: IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2006. Proceedings, Toulouse, vol. 3 (2006)
19. Wang, A., Chandrakasan, A.: A 180-mv subthreshold FFT processor using a minimum energy design methodology. IEEE Journal of Solid-State Circuits 40(1), 310–319 (2005)
20. Yamaoka, M., Maeda, N., Shinozaki, Y., Shimazaki, Y., Nii, K., Shimada, S., Yanagisawa, K., Kawahara, T.: Low-power embedded SRAM modules with expanded margins for writing. In: Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International, pp. 480–611 (2005)

Predictive Thermal Management for Chip Multiprocessors Using Co-designed Virtual Machines

Omer Khan and Sandip Kundu

Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA USA
{okhan,kundu}@ecs.umass.edu

Abstract. The sustained push for performance, transistor count, and instruction level parallelism has reached a point where chip level power density issues are at the forefront of design constraints. Many high performance computing platforms are integrating several homogeneous or heterogeneous processing cores on the same die to fit small form factors. Due to design limitations of using expensive cooling solutions, complex chip multiprocessors require an architectural solution to mitigate thermal problems. Many of the proposed systems deploy DVFS to address thermal emergencies, either within an operating system or hardware. These techniques have certain limitations in terms of response lag, scalability, cost or being reactive. In this paper, we present an alternative thermal management system to address these limitations, based on co-designed virtual machines concept. The proposed scheme delivers localized and preemptive response to thermal events, adapts well to multi-core and multi-threading environment, while delivering maximum performance under thermal stress.

Keywords: Dynamic Thermal Management (DTM), Dynamic Voltage and Frequency Scaling (DVFS), Virtual Thermal Manager (VTM).

1 Introduction

Power density problems in today's microprocessors have become a first-order constraint at run-time. Hotspots can lead to circuit malfunction or complete system breakdown. As power density has been increasing with the technology trends, downscaling of supply voltage and innovations in packaging and cooling techniques to dissipate heat have lagged significantly due to design and cost constraints. These problems are further exacerbated for small and restricted form factors.

Ideally, a thermal management solution is expected to push the design to its thermal limits, while delivering optimal system performance and throughput. As temperature is well correlated to the application behavior, it is desirable to have insight into process or thread information to guide thermal management in addition to physical triggers like distributed thermal sensors. Avoiding global trigger and response is another key requirement to ensuring scalable thermal solution for future many-core era. In order to tune for best performance at target temperature, thermal solutions need to deliver predictive response to thermal events, while keeping the response time and cost overheads low.

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 293–307, 2009.

There is an agreed hardware, software framework for power and thermal management through the ACPI framework [1]. When temperature measurements are detected and fed back to the operating system, temporal and spatial thermal aware techniques are engaged to eliminate thermal emergencies. While this has been shown to be effective in many situations, ACPI framework is far from perfect. Following are some of the shortcomings of the ACPI framework:

Current Management techniques are reactive with large response times: On-die droop and thermal sensors are in wide use today. These sensors have inaccuracy problems, which coupled with long system latencies have a detrimental effect on sense-and-react systems. For example, a computer system takes 100s of micro-seconds to adjust clock frequency and power supply voltage [2]. Additionally, a large manufacturer had a product recall for server parts in 2006 due to sensor inaccuracy [3]. As a result, sufficient guard bands must be put in place to prevent errors from creeping in during the response lag. For future technology trend projections by ITRS, as the power density rises, temperature rises will be faster, voltage and frequency response times that are gated by decoupling capacitor size and PLL lock times will remain similar, and therefore, a greater guard band has to be used [4].

Many-core Problems: In a sea-of-core design, power and thermal management is even more problematic. In POWER6 design, there are 24 temperature sensors [5]. If any of these sensors trigger, the response is global. The problem becomes more challenging when there are 100 or 1000 sensors. In that scenario, it is possible that some sensors trigger with alarming frequency. This will cause a processor to operate mostly at low performance mode. To slow down only one core, it must have its own clock distribution network. A sea-of-cores with each core having its own PLL and a private clock distribution network is a non-trivial design challenge from floor-planning and physical design point of view. These are some of the critical challenges for DFS in the future.

If one core is slowed down, the voltage (DVS) for this core cannot be reduced unless it is in a separate power island. If every core has a separate power island, there will be separate connections between external Voltage Regulator Module (VRM) and the chip, leading to congestion at the board level. Consequently, each core will have its own external decoupling capacitor leading to greater power supply noise inside each core. These issues raise the cost of implementing DVS, while the effectiveness of DVS gets reduced. The effectiveness of DVS gets further eroded in 45nm technology, where the power supply voltage is proposed to be 0.9V. For the SRAM bits to work properly, a minimum of 0.7V is needed, reducing the range of supply voltages [4].

In order to address these issues, we propose to unify the thermal monitoring and response management under a common system level framework. Some of the objectives of our proposed framework are:

Scalable thermal management: Insulating thermal management from the Operating System (OS) enables a scalable system level solution. In general, it is not a good idea to involve the OS for thermal management because that requires OS changes as the processor design evolves.

Distributed temperature monitoring: As the chips become larger and feature multiple cores, a targeted response to temperature events become necessary. A global response penalizes all threads across all cores. This is not desired.

Action based on rate of change of temperature: A major benefit of the gradient approach is that thermal emergencies can be intercepted before they occur. This allows a smaller safety margin, also known as temperature guard band. Further, sensors have inaccuracies due to process variation. Multipoint measurements are generally more accurate than single reading. This can potentially alleviate sensor inaccuracy issues.

Low response latency: Coupling predictive actions with low latency fine-grain responses can allow a system to operate closer to its temperature limit.

Application Adaptation: A tight correlation exists between temperature and application behavior. Adapting to each thread's thermal demands can optimize system performance, while keeping the cores thermally saturated.

The proposed solution is based on virtualizing the thermal management system and satisfies all of the objectives stated above.

The rest of the paper is organized as follows: In section 2, we discuss the previous work and provide motivation for our proposed framework. In section 3, we describe our proposed thermal management architecture. Section 4 discusses experimental methodology and Section 5 results and analysis. We conclude in section 6.

2 Related Work

Our approach tackles thermal management in a unified hardware/software framework. One of the first hardware, software co-design approach of dynamically managing temperature control was presented in the DEETM framework by Huang et al. [6]. Borkar identified that thermal packaging costs will increase sharply and estimated that exceeding 35-40W, thermal packaging increases the total cost per chip by $1/W [7]. Dynamic thermal management (DTM) techniques have been proposed to alleviate the thermal packaging costs by enabling the design for temperature less than the peak and use *reactive* DTM to tackle the rare case when temperature limits are approached [8]. The response mechanism initiated by DTM is typically accompanied by degradation in the performance of the chip and persists until normal system operation is resumed. DTM is the philosophy behind Intel, AMD and Transmeta microprocessor's thermal design with support for varying levels of operation and fine-grained frequency and voltage scaling [9]. Skadron et al. [10] proposed the use of control theory algorithms for DTM, using fetch gating and migrating computation as their action mechanism. Brooks et al. [8] proposed several localized reactive mechanisms – I-cache throttling, decode throttling and speculation control. They also identify dynamic as well as static triggers for these mechanisms. Dynamic triggers may be based on sensors, activity counters or dynamic profiling, whereas, static trigger may be based on compiler optimizations that estimate high-power code segments and insert instructions specifically for DTM trigger.

Rohu and Smith [11] present a software technique that allows the operating system to control CPU activity on a per-application basis. Temperature is regularly sampled and when it a reaches dangerous level, the application (or "hot" process) responsible is slowed down. This technique is shown to be superior to throttling as it does not affect slow processes. Srinivasan and Adve [12] proposed a *predictive* DTM

algorithm targeted at multimedia applications. They intended to show that predictive combination of architecture adaptation and DVS performs the best across a broad range of applications and thermal limits. Shayesteh et al. [13], Powell et al. [14], and Michaud et al. [15] investigate thread/activity migration via the Operating System as a means of controlling the thermal profile of the chip. They explore the use of swapping applications between multiple cores when a given core exceeds a thermal threshold.

2.1 Proposed Framework

We present the idea of a thermal monitoring and response management scheme under a common system level virtual framework. The core requirements of this manager are to sense the impact of temperature on various compute structures in a hardware platform, and subsequently respond by reconfiguring the platform such that the system operates at maximum performance without exceeding its thermal boundary. Pure hardware implementation of thermal manager is expensive and lacks flexibility in a typical system. On the other hand, pure software based approach needs instrumentation capabilities to tackle the requirements of managing the low level communication with the hardware platform. Additionally, operating system based implementation lacks flexibility due to strict interface abstractions to the hardware platform. These constraints drive us towards proposing a scheme which is minimally invasive to system hardware and software abstraction layers.

3 Thermal Management via Co-designed Virtual Machines

Our proposed scheme has both hardware and software components, as shown in Figure 1. The hardware component consists of thermal sensors that are strategically distributed throughout the chip. Additionally, the platform provides reconfiguration capabilities for localized throttling of resources such as queues, buffers and tables, as well as the ability to reduce the width of the major components of the processing elements such as, fetch, issue and retirement units. Finally, the hardware platform provides support for virtualization features like expanded isolation, and mechanisms for quick thread migration capabilities.

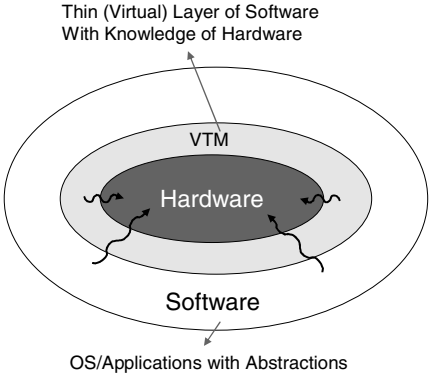


Fig. 1. Virtual Thermal Manager's System View

The software component of our solution is the thermal management software that runs natively as a guest privileged process on the hardware platform. We assume a thin Virtual Machine Monitor (VMM) running underneath the OS, which is primarily used to enter and exit, as well as pass specific thread information to the Virtual Thermal Manager (VTM) [16]. VTM software resides in the physical memory that is concealed from all conventional software including the OS. VTM software maintains temperature history tables for live threads in the system. Based on the temperature history, thread specific behavior, and the distributed thermal sensors, the VTM predicts the thermal mapping for the next epoch of computation. When VTM predicts a potential for thermal hotspots, it takes an intelligent and preemptive measure by reconfiguring the hardware platform. VTM considers the performance and thermal tradeoffs, and provides adjustments for sustained performance levels at target temperature. VTM software is akin to hypervisor that is commercially available [17].

3.1 VTM Architecture Framework

A detailed system's view of the VTM architecture is shown in Figure 2. The operating system passes information about the current and the next thread to the VTM via VMM, based on its scheduling decisions. This is conceivable with the efficient hardware and software support for virtualization. The VMM also maintains a VTM timer that is setup on every VTM exit. This timer is adjusted by the VTM to adapt its sampling to the thermal requirements. The hardware is assumed to have thermal sensors distributed across the platform. These sensors are assumed to register their readings periodically with a thermal controller. The thermal controller can interrupt to invoke the VTM in case of thermal emergency.

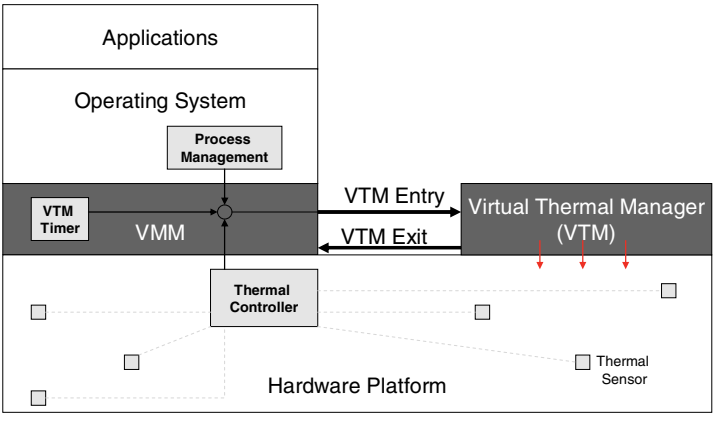


Fig. 2. VTM Interfaces and Interactions within the System

When VTM is active, it has the highest privileged access to the hardware platform. Once VTM software completes its work to determine and setup the thermal management actions, it exits via the VMM and passes control back to the OS. As a result, our approach delivers a hardware-software co-designed solution that assists the hardware to dynamically adjust to tackle the thermal concerns.

3.2 VTM Software Details

The main data structures maintained by the VTM software are Thread-to-Core Mapping Table (TCMT) and Temperature History Table (THT). TCMT maintains the thread-to-core mapping of live software threads in the system. The purpose of this table is to keep track of thread mapping, use this information to assist with thread migration and also inform the OS of such actions. TCMT ensures a programmable pre-defined time period between thread migrations, which is fixed at 10ms for this study.

THT has an entry for each live thread running in the system. For each thread entry, the THT has an entry for each distributed sensor in the hardware platform. At the lowest level, each THT sensor entry maintains data for the last two temperatures, a 2-bit saturating counter (THC) that tracks the temperature history for each live thread, running average temperature on a per thread basis, and the last DTM action determined by the VTM. Finally, VTM maintains memory mapped temperature registers, which are sampled and updated regularly by the thermal sensors. The high level software flow for the VTM is presented in Figure 3.

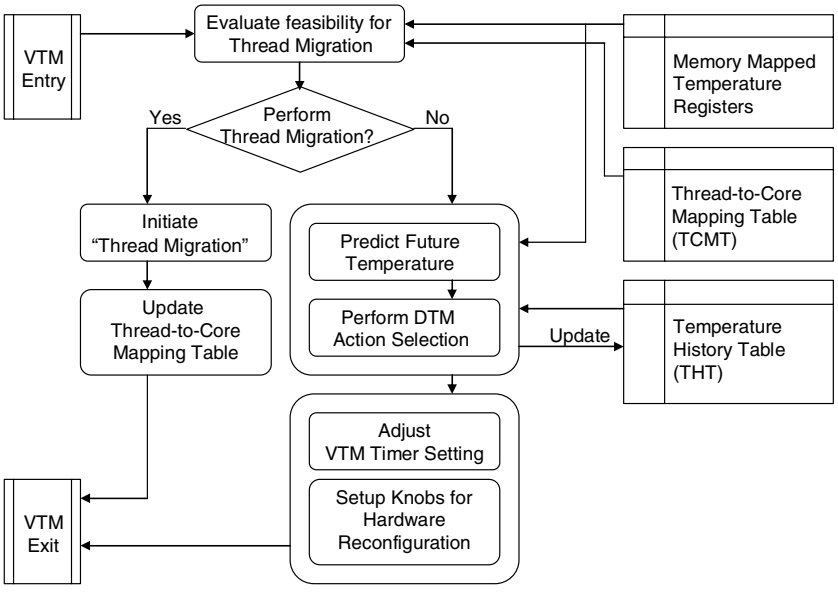


Fig. 3. VTM Software Flow

On each VTM invocation, and for each distributed sensor, the VTM reads the TCMT, THT, and memory mapped temperature registers for the current and next thread. In case of a multi-core, the temperature delta between thermal sensors on equivalent structures per logical core is evaluated for a possible thread migration. If two cores show thermal imbalance greater than a pre-defined threshold, thread migration is initiated followed by an update of TCMT with the new thread-to-core mappings. Subsequently VTM exits and gives control back to the OS.

In case when thread migration is evaluated to be infeasible, the VTM software takes a fine-grain approach to thermal management. First, the current thermal mapping of each sensor is evaluated to predict future temperature based on the current sensor reading, the last two temperature readings, as well as THC status and average thread temperature. This future thermal mapping is fed into a lookup based DTM action selection process, which also takes into account the previous DTM action before setting the next DTM action. When the future DTM action is determined for each sensor, a union of these actions is used to setup the knobs for reconfiguring the hardware platform. The next DTM action also appropriately sets the next VTM invocation delay followed by an update to the THT table and VTM exits.

The 2-bit saturating temperature history counter (THC) is updated based on comparison between the last two temperature readings. The primary purpose of THC is to capture the long term temperature behavior of each live thread and use it along-with the last few readings and average temperature to predict the future temperature. The counter's update flow is shown in Figure 4. Each process is initialized at neutral for each sensor's THC.

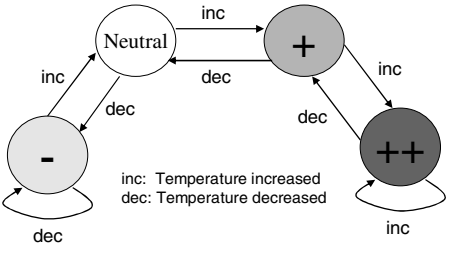


Fig. 4. THT's Temperature History Counter (THC)

We use a linear approximation model to project the temperature for the next thread. For each sensor, the projected temperature can be calculated by:

$$\text{ProjectedTemp}_{\text{Sensor\#}} = \text{CurrentTemp}_{\text{Sensor\#}} + A \quad (1)$$

The $\text{CurrentTemp}_{\text{Sensor\#}}$ is the current temperature status of a particular sensor and A is the projected increase or decrease in temperature for that sensor when the next thread will run on the platform. The calculation for A is presented below.

VTM reads the THT for the process to be run next. The THT provides the last two temperature readings for each sensor and information on the status of the associated history Counter (THC). THC state is used to select A based on equation (2).

$$A = \text{Delta}(\textit{Last Two Temps}) * \text{Constant}(\textit{based on THC \& Last Two Temps}) \quad (2)$$

where, Constant values are approximated using profiled thermal data.

3.3 DTM Action Selection

Once the projected temperature is calculated, it is used to determine the DTM action for each unit mapped to the sensor. This DTM action is calculated based on the increase or decrease of temperature, average temperature, and the last DTM action for

the thread to be run next. The selection of DTM actions is dependent on the thermal status of each sensor as well as future application demands. Our analysis of workloads indicates that a targeted, but gradual increase or decrease in DTM action severity levels yields best results in terms of thermal management. Coarse grain adjustments of DTM actions may result in an underperforming system. In our proposed system we rank DTM actions in their severity level as shown in Figure 5.

Severity ID	Issue Throttling	Issue Width	Retire Width	Speculation
0	1/1	4	4	100%
1	1/1	4	4	75%
2	1/1	2	2	75%
3	1/1	2	2	50%
4	1/1	1	1	50%
5	1/2	1	1	25%
6	1/3	1	1	25%
7	1/4	1	1	25%
8	1/5	1	1	25%
10	TCMT: Thread 1 on Core 0; Thread 2 on Core 1			
11	TCMT: Thread 2 on Core 0; Thread 1 on Core 1			

Fig. 5. DTM Actions in increasing thermal severity

Lower ID indicates a lower severity level. As the primary focus of our proposed scheme is to present a thermal management framework, the DTM actions considered here are limited, but enough for the demands of our workloads. We ran sensitivity analysis for our SPEC2000 benchmarks to determine the impact of these setting on various micro-architecture blocks. This profiled data is stored in a programmable read-only table and made accessible to the VTM. Our approach to DTM actions is to initially narrow the processor pipeline to approach a single-issue machine and then use issue throttling to further penalize the processor. Issue throttling of x/y indicates that the processor will operate at full capacity for x cycles and after every x cycles, stall for y-x cycles. Thread migration schemes for a dual-core are considered and the two possible TCMT settings are allocated Severity IDs of 10 and 11.

4 Experimental Methodology

In this section we discuss our simulation environment. We use our modified version of SESC cycle-level MIPS simulator for developing the VTM architecture framework [18]. For modeling dynamic power, SESC implements a version of Wattch [19] and Cacti [20] power estimation tools. We have extended SESC to dynamically invoke HotSpot temperature modeling tool [21]. We have also extended SESC to support DVFS. SESC supports chip multiprocessors within SMP paradigm and provides process scheduling for multi-threaded and multi-programmed workloads. We have extended SESC with context scheduler routines to manage and spawn multiple threads dynamically. This allows us to model the VTM architecture within SESC.

We use a single and a dual-core processor as our example system platform. Each core is assumed to be an aggressive pipeline with redundancy and speculation support. A temperature sensor is placed at each major micro-architecture block, allowing us to model distributed sensor architecture. System parameters used are shown in Figure 6. We assume a low cost cooling package for this study, with maximum tolerable temperature of 85°C.

Core parameters	
Fetch, Issue, Retire Width	**6, 4, 4**
L1	**64KB 4-way I & D, 2 cycles, LRU**
L2	**2M 8-way shared, 10 cycles, LRU**
ROB Size, LSQ	**152, 64**
Off-chip memory latency	**200 cycles (100 ns @ 2 GHz)**

Hotspot parameters	
Ambient Temperature	**45°C**
Package Thermal Resistance	**0.8 K/W**
Die Thickness	**0.5 mm**
Maximum Temperature	**85°C**
Initial Temperature	**76°C**
Temperature Sampling Interval	**10,000 cycles**

Fig. 6. System Parameters

We assume 65nm technology with chip wide Vdd of 1.1V, and frequency of 2.0 GHz. For comparison with VTM, we use chip level DVFS as the DTM response with Vdd of 0.9V and frequency at 800 MHz. Under DVFS, we assume sensor inaccuracy of ±3°C [2], a response latency of 100us, and an upper/lower temperature threshold of 82°C & 81.5°C respectively. Comparatively, we assume that our approach will result in better sensor accuracy, as multipoint temperature readings used to project future temperature statistically provides more accurate readings [10]. VTM is sampled every 1 to 50ms in our setup, with an estimated entry to exit delay penalty of 2000 clock cycles for each invocation. For each thread migration an additional 20us penalty is assumed for flushing core pipeline and transferring architecture state. We choose 0.5°C temperature threshold to initiate a thread migration, which yields best results in terms of temperature variations. VTM temperature threshold is set to 83°C.

For analysis, we choose SPEC2000 benchmarks. The choice of benchmarks is primarily based on thermal behavior of the workload with MCF being *cold*, GCC, PARSER, AMMP being *moderate* and EQUAKE, ART, BZIP2 being *hot*. We also run a multi-threaded grouping of these benchmarks on the single and dual-core platforms. Each thread is fast forwarded 2 billion instructions, followed by HotSpot initialization. This ensures that the cores as well as HotSpot and VTM history tables get sufficient warm up.

For each benchmark, three simulations are conducted. First simulation is without any DTM support and we categorize it as *no-DTM*. Second simulation is with *DVFS* as the DTM response and the trigger is based on the worst case sensor reading. Third simulation is with the VTM framework. Each simulation is run for 1 billion instructions and the performance of each simulation is evaluated based on Millions of Instruction per Second (MIPS). For dual-core, each simulation is run until the average instruction count seen by the two cores reaches 1 billion.

5 Results and Analysis

In our simulation runs, the integer register file is the hottest unit, followed by the integer window and the load queue. The data is shown for the worst-case unit in each core. The x-axis shows the relative performance in terms of throughput. Initially, the application runs in full performance mode. But as the thermal sensors observe rising temperatures, VTM adapts to workload demands to prevent exceeding thermal limits.

5.1 Analysis of Single Core

Figure 7 and Figure 8 show the three simulation runs for a selective number of benchmarks on a single-core. VTM is adjusting DTM actions for the best possible performance, while keeping the core within its thermal envelope. The history of thread activity and the temperature rate of change cooperatively provide insight for VTM to adjust in a smart and predictive manner rather than abruptly, as seen in DVFS simulations. Figure 7 shows a multi-threaded workload with two threads running in a time-multiplexed fashion on a single-core. The thermal behavior of the platform changes when such cooperative scheduling scenarios are considered. The VTM software adapts to each thread's thermal behavior, subsequently predicting and managing the future trend on a per thread granularity.

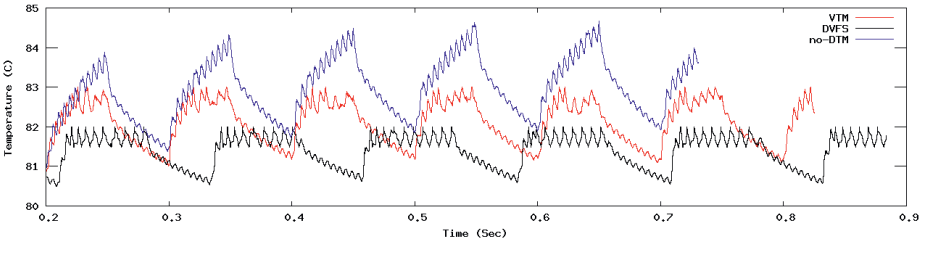


Fig. 7. ART-MCF comparisons for IntReg in single-core

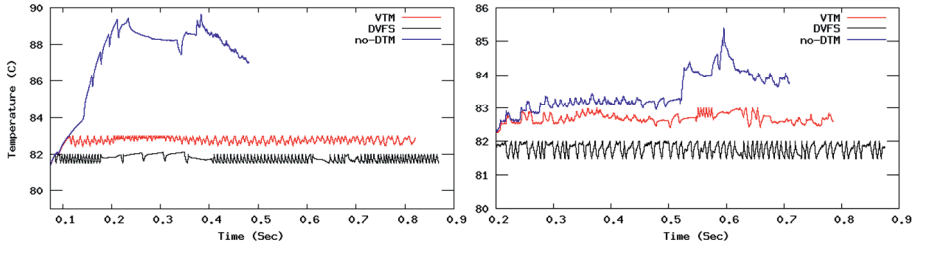


Fig. 8. BZIP2 and GCC comparisons for IntReg in single-core

GCC plot in Figure 8 identifies the scenario when sudden temperature spikes can cause instant and short lived thermal alarms. Although VTM is designed with variable invocation to gracefully handle such situations in software, it may be unavoidable in all cases. Our solution, which adds an extra hardware interrupt to invoke VTM when some upper temperature threshold is detected, gracefully handles thermal spikes.

5.2 Analysis of Dual Core

Figure 9 and Figure 10 show the three simulation runs for a selective number of benchmarks on a dual-core. VTM manages both threads cooperatively to push performance and thermal behavior of each core in the platform. Figure 9 shows two hot threads running simultaneously on both cores in the system. Even with a 60% degraded performance of DVFS, the core 1 exceeds the temperature threshold under a global DVFS scheme. Such scenarios can result in uncontrolled temperature spikes causing major damage to the system. On the other hand, VTM uses thread migration to primarily push the cores to their thermal limits and then use fine-grain predictive hardware reconfiguration to adjust DTM actions. In this scenario, when both threads are hot, fine-grain DTM actions are used to control the temperature of the dual-core.

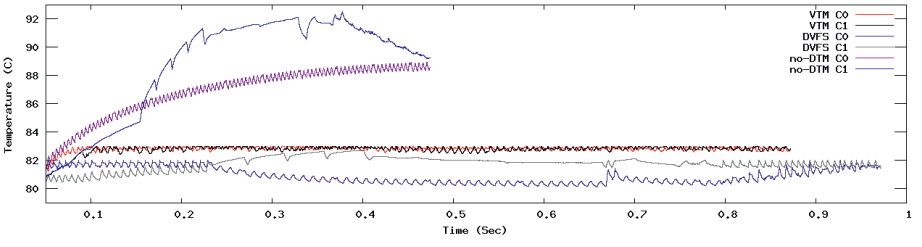


Fig. 9. ART-BZIP2 comparisons for IntReg in dual-core

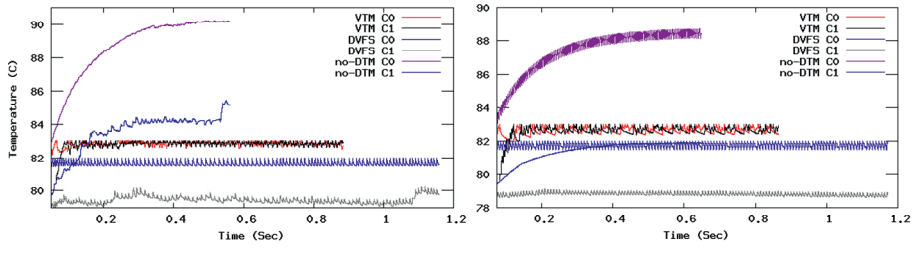


Fig. 10. EQUAKE-GCC (L) & MCF-ART (R) IntReg in dual-core

Figure 10 shows a couple of workloads with a mix of hot and moderate/cold threads. If chip level DVFS is employed in such scenarios, one of the threads underperforms due to the thermal imbalance, thus resulting in an unnecessary performance loss. Although core (or component) level DVFS is possible, the cost and scalability effect of such a scheme makes it unfeasible in future technology nodes. As an alternative, VTM provides an architectural solution to manage such workload scenarios. When a hot and cold thread is used to run on the cores, thread migration is shown to provide best performance, while keeping the chip thermally saturated. Figure 11 shows that VTM uses thread migration coupled with fine-grain DTM actions to effectively and consistently manage temperature behavior on the two cores.

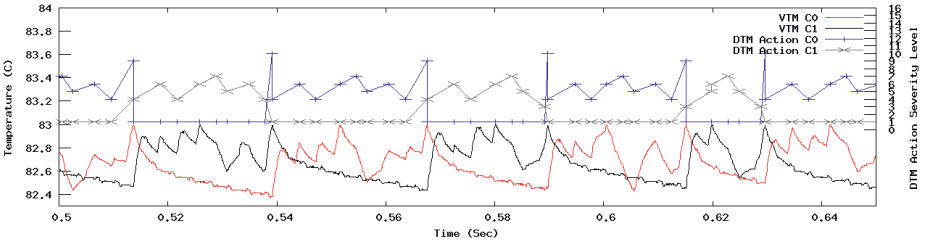


Fig. 11. ART-MCF DTM Actions over time

5.3 Analysis of Localized Responses and Thread Adaptation

Figure 12 and Figure 13 show the temperature behavior of the IntReg sensor along with the corresponding DTM actions initiated by the VTM. A detailed analysis of DTM actions shows that VTM gradually adapts to the workload behavior, continuously evaluating system performance tradeoff with the thermal limitations. Figure 13 shows that VTM effectively handles each thread independently, so for ART, activity is high and thus DTM adjustments are more fine-grain to, whereas, for MCF, activity is low and thus DTM actions are adjusted accordingly. As EQUAKE-GCC workload mixes a hot and a moderate thread, the DTM actions are severe for EQUAKE and moderate for GCC.

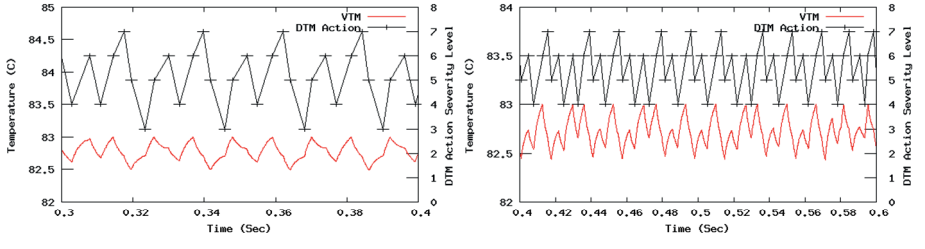


Fig. 12. EQUAKE (left) and BZIP2 (right) IntReg DTM actions

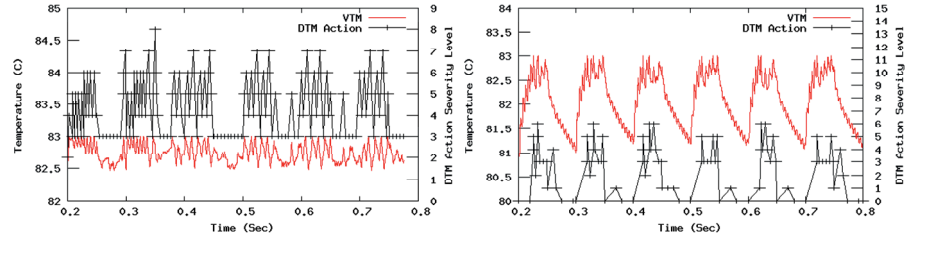


Fig. 13. EUQAKE-GCC (left) and ART-MCF (right) IntReg DTM actions

5.4 Analysis of VTM Performance

Figure 14 summarizes the performance impact of our VTM approach for thermal management compared to DVFS. Our data shows that VTM consistently delivers better performance than the DVFS counterpart. VTM over DVFS shows a minimum of 6% and maximum of 35% improvement. The average improvement over all benchmarks considered is 8% for single-core and over 32% for dual-core processors. DVFS implementation considered in this study may be considered sub-optimal due to the limited voltage/frequency levels. We consider this fair, as DVS is not predicted to scale for future technology nodes. Additionally, our analysis indicates that global DVFS is more of a problem in terms of performance loss compared to the number of available stepping. We consider core level DVFS as an unfeasible solution due to its design implementation costs and scalability limitations for many core era.

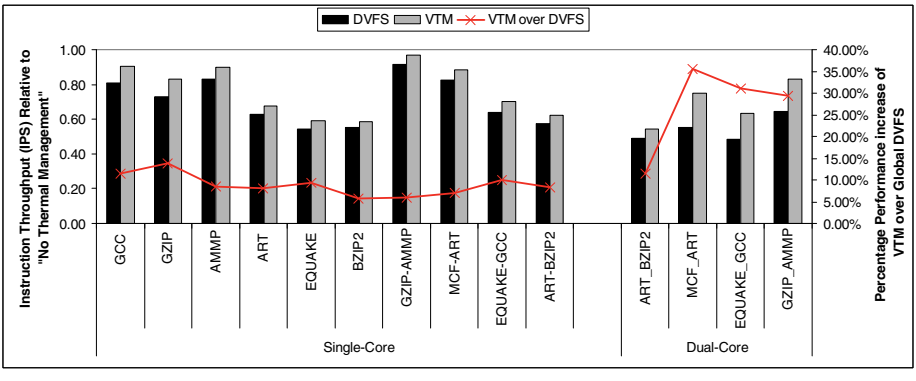


Fig. 14. Performance of Thermal Management Schemes

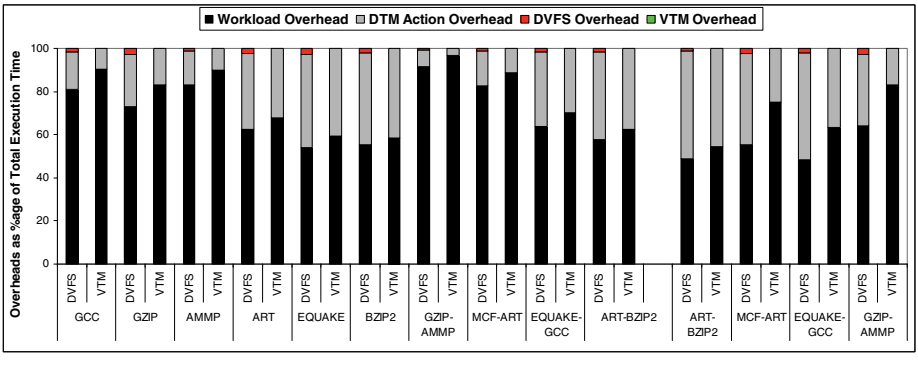


Fig. 15. Analysis of Thermal Management Overheads

Figure 15 breaks down the execution time into several overheads. The workload overhead is the necessary component consumed by executing the workload. All other overheads are related to thermal management. DTM action overhead is the time taken by DTM actions to degrade performance, but adjust the thermal behavior to match

target temperature. The DVFS overhead is the 100us overhead of every DVFS invocation, whereas, VTM overhead is the time taken by VTM to compute next DTM action. The data clearly indicates that VTM and DVFS overheads are a negligible percentage of the total execution time. Specifically, VTM overhead plays minimal role in performance degradation of the processor. This makes the case for using virtual machine as an effective mechanism for thermal management.

5.5 VTM for Reduction of Leakage Power

The proposed use of VTM to keep all cores in the CMP thermally saturated can be used to reduce the leakage power. Our data shows that some cores when running at elevated temperature with severe hot spots will result in higher leakage, while cooler cores will result in lower leakage. When VTM is deployed, all cores are thermally controlled to stay close to the pre-determined temperature levels. As the peak temperatures are reduced the relatively cooler cores become warmer. Because the leakage power is a nonlinear (exponential) function of temperature, the leakage savings incurred by lowering the maximum target temperature outweighs the increase in leakage incurred by increasing the temperature of the cooler cores. Such leakage reductions are conceivable within our proposed framework, while rest of the VTM management remains the same. We intend to study such enhancements in our future work.

6 Conclusions

We have presented a novel thermal management scheme based on using virtual machine to manage DTM action selection and scheduling. The main reason for using virtual machine is its lower overhead cost and flexibility to enable changing thermal demands of many core designs and applications. We studied VTM as a standalone software scheme with hardware assistance.

The proposed thermal management scheme adapts to each thread individually to match the computation demands with the hardware thermal profile. The response time is quick because the proposed scheme is not reliant on PLL lock time and decoupling capacitor charge/discharge time. VTM provides a scalable path to many-core era where DVFS will be limited by physical design and technology constraints. The proposed technique delivers the best performance at target temperature in a distributed thermal sensing environment.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 0649824.

References

1. Advanced Configuration and Power Interface (ACPI). Document, `http://www.acpi.info/spec.html`
2. Naffziger, S., et al.: Power and Temperature Control on a 90nm Itanium®-Family Processor. In: Proceedings of Int'l Solid State Circuits Conference (2005)

3. Revision Guide for AMD NPT Family 0Fh Processors, AMD Publication # 33610, p. 13 (October 2006)
4. International Roadmap for Semiconductor (ITRS). Document, http://public.itrs.net/
5. Sanchez, H., et al.: Thermal System Management for high performance PowerPC microprocessors. In: Proceedings of COMPCON (1997)
6. Huang, M., Renau, J., Yoo, S., Torrellas, J.: The Design of DEETM: A Framework for Dynamic Energy Efficiency and Temperature Management. Journal of Instruction-Level Parallelism 3 (2002)
7. Borkar, S.: Design Challenges of Technology Scaling. IEEE Micro (July-August 1999)
8. Brooks, D., Martonosi, M.: Dynamic Thermal Management for High-Performance Microprocessors. In: Proceedings of Int'l Symposium on High Performance Computer Architecture (2001)
9. Burd, T.D., et al.: Dynamic Voltage Scaled Microprocessor System. IEEE Journal of Solid-State Circuits (November 2000)
10. Skadron, K., et al.: Temperature-aware computer systems: Opportunities and challenges. IEEE Micro. 23, 52–61 (2003)
11. Rohou, E., Smith, M.: Dynamically managing processor temperature and power. In: 2nd Workshop on Feedback Directed Optimization (November 1999)
12. Srinivasan, J., Adve, S.V.: Predictive Dynamic Thermal Management for Multimedia Applications. In: Proceedings of Int'l Conference on Supercomputing (June 2003)
13. Kursun, E., Reinman, G., Sair, S., Shayesteh, A., Sherwood, T.: Low-overhead core swapping for thermal management. In: Falsafi, B., VijayKumar, T.N. (eds.) PACS 2004. LNCS, vol. 3471, pp. 46–60. Springer, Heidelberg (2005)
14. Powell, M., Gomma, M., Vijaykumar, T.: Heat-and-run: Leveraging SMT and CMP to Manage Power Density through the Operating System. In: Proceedings of Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (2004)
15. Michaud, P., et al.: A study of thread migration in temperature-constrained multi-cores. ACM Transactions on Architecture and Code Optimization (2007)
16. Smith, J., Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann Publication, San Francisco (2005)
17. IBM Systems Virtualization, IBM Corporation, Version 2 Release 1 (2005)
18. Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., Montesinos, P.: SESC Simulator (2005), http://sesc.sourceforge.net
19. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: A framework for architectural-level power analysis and optimizations. In: Proceedings of Int'l Symposium on Computer Architecture (2000)
20. Shivakumar, P., Jouppi, N.P.: CACTI 3.0: An integrated cache timing, power, and area model, Technical Report, Compaq (2001)
21. Skadron, K., et al.: HotSpot: Techniques for Modeling Thermal Effects at the Processor-Architecture Level. In: THERMINIC (2002)

HeDGE: Hybrid Dataflow Graph Execution in the Issue Logic

Suriya Subramanian and Kathryn S. McKinley

Department of Computer Sciences
The University of Texas at Austin
{suriya,mckinley}@cs.utexas.edu

Abstract. Exposing more instruction-level parallelism in out-of-order superscalar processors requires increasing the number of dynamic in-flight instructions. However, large instruction windows increase power consumption and latency in the issue logic. We propose a design called Hybrid Dataflow Graph Execution (HeDGE) for conventional Instruction Set Architectures (ISAs). HeDGE explicitly maintains dependences between instructions in the issue window by modifying the issue, register renaming, and wakeup logic. The HeDGE wakeup logic notifies only consumer instructions when data values arrive. Explicit consumer encoding naturally leads to the use of Random Access Memory (RAM) instead of Content Addressable Memory (CAM) needed for broadcast. HeDGE is distinguished from prior approaches in part because it dynamically inserts forwarding instructions. Although these additional instructions degrade performance by an average of 3 to 17% for SPEC C and Fortran benchmarks and 1.5% to 8% for DaCapo Java benchmarks, they enable energy efficient execution in large instruction windows. The HeDGE RAM-based instruction window consumes on average 98% less energy than a conventional CAM as modeled in CACTI for 70nm technology. In conventional designs, this structure contributes 7 to 20% to total energy consumption. HeDGE allows us to achieve power and energy gains by using RAMs in the issue logic while maintaining a conventional instruction set.

1 Introduction

To attain high performance, superscalar processors seek to exploit Instruction Level Parallelism (ILP) with large instruction windows and dynamic scheduling algorithms. The instruction issue logic is thus a key component in their design.

Current instruction window designs use a monolithic Content Addressable Memory (CAM) because it implements broadcast efficiently for instruction wakeup. Unfortunately, CAM structures scale poorly with respect to latency and power. Increasing the size of CAMs to expose more ILP forces a tradeoff between ILP and the clock period; larger CAMs consume disproportionately more power, which forces a tradeoff between power and performance. This paper seeks scalable instruction issue logic to attain energy efficiency together with high performance.

Our solution replaces broadcasts in the issue logic with direct communication. Current issue logic performs broadcast when producer instructions complete, notifying dependent instructions waiting in the issue window. Prior work shows that there are few

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 308–323, 2009.

such consumers within the window [6]. We find that 94 to 96% of instructions produce a result for two or fewer consumers in windows ranging from 64 to 512 instructions. This observation motivates a design that uses dataflow encoding of dependent instructions in the window, i.e., *direct instruction communication* between producers and consumers, instead of a broadcast based on physical register names. Our design dynamically identifies and encodes consuming instructions during the register rename stage. When an instruction produces its result, the wakeup logic identifies consumers and marks them ready. Eliminating the need for a broadcast leads to an instruction window implementation that uses a Random Access Memory (RAM) instead of a CAM. RAMs offer two significant advantages over CAMs: they consume significantly less energy per access and have a faster access time.

We call our design Hybrid Dataflow Graph Execution (HeDGE) because it takes an intermediate point between conventional superscalars and dataflow ISAs, such as WaveScalar [24] and TRIPS [18]. HeDGE requires no changes to a conventional ISA. It dynamically converts dependences specified with register names in the ISA as follows. When a consumer enters the window, HeDGE register renaming adds the consumer to a wakeup list for the producer. This logic generates a dataflow encoding, but only for instructions in the issue window. HeDGE implements the wakeup list by adding target fields to the reservation stations. When the number of consumers exceeds the number of target fields, HeDGE introduces forwarding instructions. Dynamically inserting forwarding instructions differentiates HeDGE from prior approaches to direct instruction communication in conventional designs, which stall the pipeline [26], or continue to use some associative logic for the instruction window [13,21], or sacrifice more ILP to track consumers [19].

The contribution of this paper is the demonstration and design of a power efficient instruction window that supports many in-flight instructions by using a more scalable hardware structure. We measure HeDGE in a cycle-accurate simulator on SPEC CPU and DaCapo Java benchmarks. Given two to four target fields in the HeDGE reservation stations and a range of issue window sizes of 64 to 512, HeDGE requires 2 to 30% additional forwarding instructions on average. Although these instructions degrade performance by an average of 3 to 17%, they enable energy efficient execution in large instruction windows. Using CACTI to model RAM and CAM structures in 70nm technology, we find that the energy per access consumed by a HeDGE RAM is 98% less than a CAM. In a conventional design, prior work shows that the CAM-based instruction window contributes 7 to 20% to total energy consumption [4,10,11], and the contribution increases as a function of the window size. Assuming a conservative 10% contribution, we show that HeDGE configurations reduce total processor energy by an average of 6%. RAMs also offer faster access times, but we do not explore this benefit here. These results demonstrate the potential of HeDGE designs to improve power efficiency.

2 Related Work

This section describes related work in issue logic design that uses explicit dependence tracking, that reduces issue load, and that uses dataflow ISAs. We also provide a brief taxonomy of dependence encoding. We refer the reader to Abella et al. for a comprehensive survey of issue logic design [1].

Dependence tracking. The most closely related work seeks to use direct instruction communication to reduce the complexity of the issue logic in dynamically scheduled superscalar processors [13,19,21]. These approaches explicitly track register dependences between instructions and completely or partially avoid associative lookup during instruction wake up. Similar to HeDGE, these approaches rely on the observation that only a few dependent instructions are typically in the issue window at a time and therefore propagate result tags only to those instructions in the window. However, none of these approaches considered or modeled energy-delay benefits.

Similar to HeDGE, Önder and Gupta use a fixed fanout degree [19]. However, when the number of targets exceeds the fanout degree, they encode the chain of forwarding instructions together with the consuming operands. When an instruction executes, the hardware forwards its result to consumers and its input operands to other instructions needing the same value. Each value is forwarded on a separate cycle, whereas HeDGE inserts `MOV` instructions, and delivers all the target fields of `MOV` and other instructions at once by using additional logic.

Sato et al. use a RAM-based instruction window with a register file called the Dataflow Management Table (DMT) to keep track of dependences [21]. This scheme eliminates associative wakeup; however, they must checkpoint the DMT on every branch prediction, as the DMT might contain incorrect dependences after a branch misprediction. HeDGE instead uses the misprediction handling mechanism that already exists in a superscalar processor.

Huang et al. modify the instruction window to maintain dependence information between a producer with a single consumer within the window, and then wake up just the consumer, avoiding a broadcast [13]. If more than one consumer enters the window, the wakeup logic reverts to a conventional broadcast scheme. This hybrid design combines direct instruction wakeup and broadcast, but comes with additional complexity. HeDGE uses `MOV` instructions when there are multiple dependent consumers within the window. This design adds instruction overhead compared to Huang et al., but enables the use of RAM hardware and simplifies the instruction window design.

Reducing issue logic latency. To reduce the issue logic latency, a number of approaches perform some form of dependence-based pre-scheduling to reduce the number of instructions considered for issue every clock cycle [16,17,20]. Palacharla et al. performed an analysis of circuit delay of various structures in a superscalar processor, and showed that the wakeup and select logic is a key element of the processor's critical path [20]. They proposed the first dependence-based instruction window design where the issue queue is implemented as a set of FIFOs with only the head of the FIFOs considered for issue. Michaud and Seznec pre-schedule instructions based on dataflow order, grouping instructions based on the clock cycle at which they will issue, thereby reducing the number of instructions considered for selection [17]. Lebeck et al. identify instructions dependent on long-latency operations such as cache misses and move them to a larger buffer [16]. They move these instructions back to the issue queue when the long latency operation completes. The number of instructions in the issue queue is smaller, and thus the issue queue is faster. These approaches are orthogonal to HeDGE and can coexist with our approach.

A taxonomy of dependence encoding. The taxonomy in Table 1 classifies Von Neumann and dataflow architectures according to the way they specify dependences between instructions in the ISA and between instructions in the issue window. Conventional out-of-order superscalar processors like the Alpha 21264 [15] use a Reduced Instruction Set Computer (RISC) instruction stream that encodes dependences between instructions using register names. The initial stages of the pipeline use register renaming to eliminate write-after-read and write-after-write dependences. Read-after-write dependences between instructions within the window are specified using physical register names.

WaveScalar, TRIPS, and other dataflow machines directly encode dependences in the ISA to exploit the inherent efficiencies of dataflow execution [7,18,24]. In a dataflow ISA, the compiler must explicitly specify dependences between instructions using target instruction identifiers. The execution model maps instructions to execution units on a distributed substrate, preserving the dependence information encoded in the ISA. In WaveScalar and TRIPS, both the ISA and microarchitecture use instruction identifiers to specify dependences. HeDGE exploits some of the same efficiencies, but in the context of a conventional ISA.

Table 1. Taxonomy of dependence encoding

		Instruction Window Encoding	
		Register names	Instruction names
ISA encoding	Register names	Alpha 21264 [15], Pentium	Huang et al. [13], Sato et al. [21], HeDGE
	Instruction names	None	Dataflow machines [7], WaveScalar [24], TRIPS [18]

3 Background

This section describes a conventional superscalar pipeline, with the register renaming and instruction wakeup, to provide context and motivation for our approach.

Figure 1 depicts the pipeline stages for dynamic instruction scheduling in an out-of-order superscalar processor. The frontend of the processor (not shown) fetches, decodes, and transfers instructions to the rename stage, which keeps track of instructions by reserving reorder buffer entries, reservation stations, and physical registers. The issue stage holds instructions in reservation stations, waiting for their input operands to become available. The select logic chooses candidates for execution, from ready instructions whose input operands are all available, based on availability of execution units and other policy considerations such as age of the instruction and criticality of the instruction [8]. Instructions selected for execution, read values from the register file and execute on appropriate functional units.

Register renaming. The register renaming stage updates a Register Alias Table (RAT) that maps architectural register names to physical register names. The rename stage eliminates all write-after-write and write-after-read register dependences by mapping the write target to a unique unused physical register location. The rename logic uses this

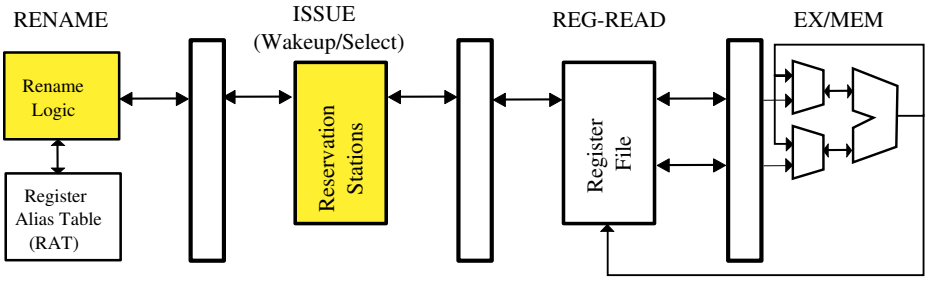
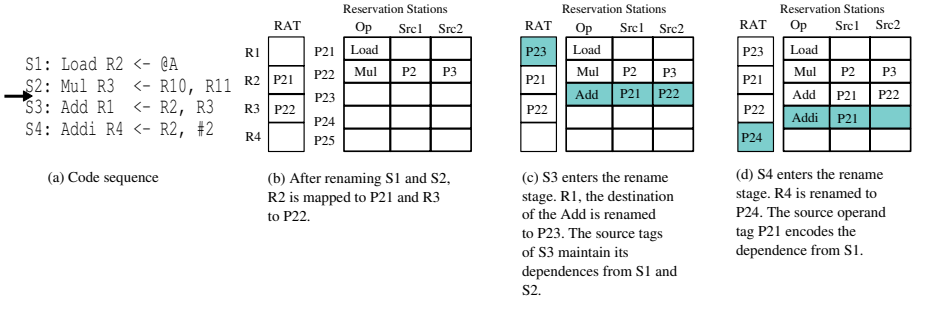


Fig. 1. Superscalar pipeline stages (HeDGE modifies the shaded parts of the pipeline)



(a) Code sequence

(b) After renaming S1 and S2, R2 is mapped to P21 and R3 to P22.

(c) S3 enters the rename stage. R1, the destination of the Add is renamed to P23. The source tags of S3 maintain its dependences from S1 and S2.

(d) S4 enters the rename stage. R4 is renamed to P24. The source operand tag P21 encodes the dependence from S1.

Fig. 2. Conventional superscalar pipeline register renaming example

physical register name to satisfy any subsequent reads from the original architectural register. Instructions speculatively issue and write to physical register storage, with the value becoming part of the architectural state only after the instruction commits. The wakeup logic uses physical register tags to check availability of operands that were not ready during register renaming.

Register renaming example. Figure 2 walks through a simple code sequence, showing the contents of the RAT and reservation station entries at each clock cycle assuming a 1-wide pipeline. Each diagram shows the RAT indexed from `R1` through `R4`, and reservation stations indexed by physical register names `P21` through `P25`. The shaded entries indicate those written in the current clock cycle. Figure 2(b) shows the state after renaming the `Load` and `Mul` instructions. Physical register `P21` maps to `R2`, the destination architectural register of the `Load`. Similarly, `R3` maps to `P22`. S3, the `Add` instruction enters the rename stage next. The rename logic allocates a new physical register `P23` to the output register `R1`. The source physical register tags respectively contain `P21` and `P22`, and maintain the read-after-write dependences from the `Load` and `Mul` instructions as shown in Figure 2(c). Similarly in Figure 2(d), the `Addi` enters the rename stage, and its output register `R4` is mapped to `P24`. Its source tag P21 encodes the dependence on S1.

Instruction wakeup. The wakeup logic is a significant source of complexity for out-of-order superscalar processors. The issue stage uses the source physical register tags set

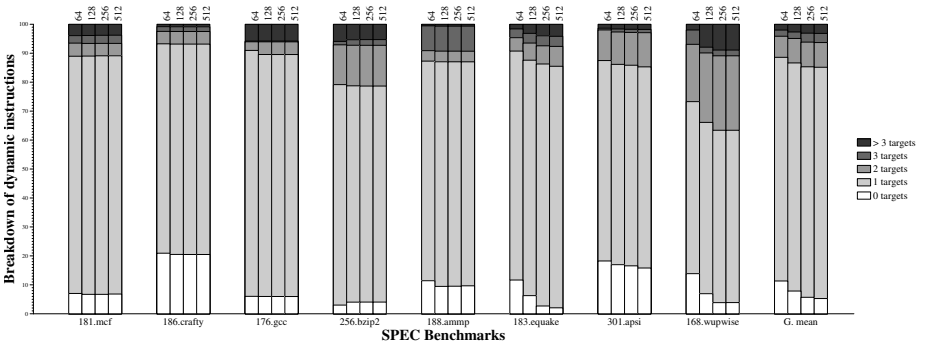


Fig. 3. Breakdown of dynamic instructions based on the number of dependent instructions within the window for instruction window sizes ranging from 64 to 512

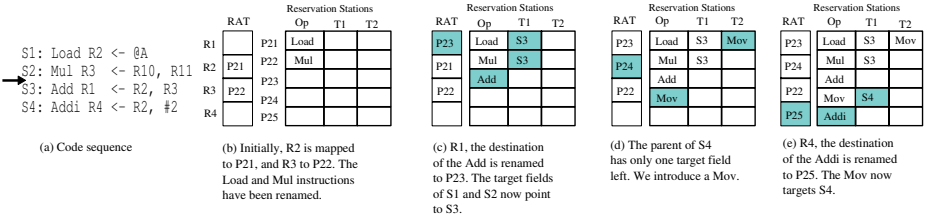
by the rename stage to wake up dependent instructions waiting in reservation stations. Just before an instruction finishes executing, it broadcasts its destination physical register tag to a common result tag bus. Waiting instructions snoop this bus and notify the *select logic* when all their operands are available. The select logic chooses candidate instructions for execution based on some heuristic. The wakeup and select logic of the instruction window is a key component in the critical path for an out-of-order superscalar processor [20]. The tag comparisons performed every cycle are a main source of complexity [20] and power dissipation in the instruction window [4,9].

4 Hybrid Dataflow Graph Execution (HeDGE)

To show the potential of direct instruction communication in the instruction window, Figure 3 shows the dynamic distribution of this communication. We measured performance on 17 of 21 C and Fortran SPEC CPU 2000 benchmarks [22] and 7 of 11 Java programs from the DaCapo benchmark suite (version `dacapo-2006-10`) [3] on which our baseline simulator currently works. We simulated the SPEC programs using the SimpleScalar 3.0 tool suite [5] for the Alpha ISA to simulate a 4-wide dynamically scheduled superscalar processor with varying window sizes, and the DaCapo programs running on JikesRVM using Dynamic SimpleScalar [14] for the PowerPC ISA. Due to space constraints, we present geometric means and representative results. A companion technical report presents all benchmark results [23]. The figure breaks down dynamic instructions based on the number of dependent instructions within the window. These results show that 94 to 96% of instructions produce a result for two or fewer consumers in windows ranging from 64 to 512 instructions, promising an efficient alternative.

4.1 Design

We leverage this observation with an instruction window design which explicitly keeps track of dependent instructions by adding target fields to the reservation stations. In a HeDGE window, producer instructions explicitly encode dependent consumer instructions, like in a dataflow machine. The HeDGE design only requires changes to the



(a) Code sequence

(b) Initially, R2 is mapped to P21, and R3 to P22. The Load and Mul instructions have been renamed.

(c) R1, the destination of the Add is renamed to P23. The target fields of S1 and S2 now point to S3.

(d) The parent of S4 has only one target field left. We introduce a Mov.

(e) R4, the destination of the Addi is renamed to P25. The Mov now targets S4.

Fig. 4. HeDGE register renaming example

rename and issue stages in an out-of-order pipeline. These stages are highlighted in Figure 1. We first summarize the key components in our design and then describe each component in detail.

When a consumer enters the instruction window, HeDGE first translates the architectural registers to physical register names, and then uses the physical name to dynamically identify any producers already in the window. It then adds the consumers to the producers' list of targets, stored in reservation stations. If the target fields are exhausted, HeDGE inserts `MOV` instructions. The `MOV` instructions and their target fields fan out values to multiple consumers when necessary. This process explicitly encodes read-after-write register dependences between instructions. This additional complexity in the rename stage results in simpler wakeup logic. The HeDGE wakeup logic looks up dependent instructions in the target fields of the reservation stations and sends the result tag only to these consumers.

Rename stage. Like in a conventional pipeline, HeDGE's rename stage maps architectural registers to physical register names for every instruction. In addition, it looks up an instruction's physical register operands in the RAT. If there is no entry for an operand, the rename stage marks the input operand as ready. Otherwise, an entry in the RAT provides the identifier for the producer. HeDGE adds this consumer to the producer's list of dependent instructions. This step dynamically encodes read-after-write dependences. Each reservation station has a small, fixed number of target fields, and there may be more consumer instructions within the window than target fields. Instead of stalling the pipeline [26], HeDGE introduces `MOV` instructions into the pipeline to track multiple consumers. We describe and use a simple algorithm that inserts a linear chain of `MOV` instructions. Although we do not evaluate it here, the renaming logic could create a tree of `MOV` instructions to fanout values in parallel.

Register renaming example. This section illustrates how HeDGE renames registers and introduces `MOV` instructions into the pipeline with the example from the previous section. Figure 4 is similar to Figure 2 but shows reservation stations with target instruction fields instead of those for source registers.

Figure 4(b) shows the state after renaming `S1` and `S2`. Renaming assigns physical register `P21` to `R2`, the destination architectural register of the `Load`. Similarly, it assigns `R3` to `P22`. Next, the `Add` instruction enters the rename stage. The rename stage allocates a new physical register `P23` to output register `R1`. For each input operand, it looks up the producer instruction identifier in the RAT. The rename stage then adds the current

instruction identifier to the target field of each producer. Figure 4(c) shows these updates by shading the new RAT and reservation station entries.

When S4, the Addi instruction, enters the rename stage, its producer, S1, has only one target field left. To accommodate more potential future consumers of S1, the rename stage inserts a MOV instruction and puts the MOV instructions identifier in the producer's target field. To make the MOV instruction the new producer of R2, it changes the RAT entry for R2 to P24. Figure 4(d) shows this intermediate step. This process is semantically equivalent to adding Mov R2 <- R2 at this point in the program. The MOV introduces a bubble in the pipeline. In the next cycle, the rename logic inserts S4's instruction identifier in the MOV's target field and inserts S4 in the reservation station, as shown in Figure 4(e).

Instruction wakeup. We now describe the instruction wakeup logic. The key difference between a conventional out-of-order processor and HeDGE lies in how producer instructions communicate availability of an operand to the wakeup logic. In HeDGE, the wakeup logic does not snoop the result bus for matching physical register tags. Instead, it directly notifies consumer instructions as producers complete. The wakeup logic indexes the reservation station table by the target fields of the producer instruction, notifying each consumer that an input operand is available. The select and execute logic in HeDGE is the same as a conventional processor; it chooses which of the ready instructions to schedule for execution and executes them on functional units.

4.2 Speculation with HeDGE

HeDGE supports existing misspeculation recovery mechanisms in a straightforward manner. Just like in conventional processors, branch instructions trigger a RAT checkpoint. When the hardware detects a branch misprediction, it squashes instructions along the mispredicted path. If a producer is on the mispredicted path, all its consumers must also be on the mispredicted path, and the hardware squashes all of them. If only the consumer in a dependent chain needs to be squashed, its producers' target fields become invalid. To address this problem, HeDGE stores *instruction numbers (inums)* together with consumer identifiers in the target fields, and only wakes up consumer instructions when inums match.

4.3 Design Tradoffs

This section discusses in more detail the implications of fixing the number of target fields and the consequent insertion of MOV instructions.

The number of target fields determines the number of MOV instructions HeDGE will insert; fewer target fields require more MOV instructions, but fewer ports and a simpler wakeup logic. Because the issue logic cannot predict in advance whether it will need a MOV instruction or not, each reservation station entry must have at least two target fields, one for a consumer and one for a MOV instruction to propagate the value. A design that does not dynamically insert MOV instructions or has a single target field must stall the pipeline when an instruction runs out of targets. HeDGE avoids such pipeline stalls, by reserving the last target field for MOV instructions. In the case where there is

exactly one additional consumer, the MOV is unnecessary. However, our conservative policy simplifies the logic for introducing MOV instructions and handling them in the other stages of the pipeline.

The semantics of a dynamically introduced MOV instruction are the same as a MOV instruction of the form "MOV Ra, Ra" where Ra is the architectural register name. A MOV instruction behaves just like any other MOV instruction within the pipeline, occupying reservation station slots and reducing the effective size of the window. They also reduce the effective issue and commit width of the processor. Finally, whenever a MOV instruction forwards a data value, it introduces a bubble in the pipeline. Section 5.2 quantifies these effects.

A HeDGE design must choose a sweet spot between increasing complexity to support more targets and consequently inserting fewer MOV instructions, or inserting more MOVs to reduce complexity.

5 Evaluation

This section describes our cycle-accurate and power-modeling simulation methodologies and results. To demonstrate the tradeoffs in the HeDGE design, we measure the number of forwarding instructions HeDGE introduces and their effect on total performance for a range of HeDGE configurations. We then model the power and energy characteristics of the circuits in a conventional CAM instruction window and in a HeDGE RAM instruction window structures using CACTI 4.2 [25]. We use prior research that specifies 7 to 20% of total energy consumption of a superscalar processor is due to the dynamic scheduling structures [4,10,11]. Over this range of values, we compare total power and energy-delay of HeDGE to a conventional design. We show that even with a conservative 10% contribution of the issue logic to total processor power, HeDGE configurations reduce total energy by an average of 6% for SPEC programs and 10% for DaCapo programs.

5.1 Methodology

We extend sim-outorder, a cycle-accurate simulator from the SimpleScalar 3.0 tool suite [5] for the Alpha ISA to implement HeDGE, for executing C and Fortran programs. We use Dynamic Simplescalar [14] for the PowerPC ISA for simulating Java programs running on JikesRVM [2]. The cycle-level simulator models an aggressive 4-way out-of-order superscalar microarchitecture. The simulator is execution-driven and accounts for instructions along the wrong path of a misspeculation. The memory hierarchy has two levels of caches with split L1 instruction and data caches and a unified L2 cache. HeDGE modifies the register renaming and wakeup logic to track a parent instruction's targets. We explore configurations with two, three, and four target fields in the reservation stations. Table 2 contains the simulation parameters, latencies, and branch predictor information.

We evaluate HeDGE on all the programs that successfully execute on our baseline simulators. We execute 17 of the 21 C and Fortran SPEC CPU2000 benchmarks [22]. We compiled all the SPEC benchmarks with Compaq's GEM compiler with full optimization for an Alpha 21264 machine. We used SimPoint 3.0 [12] to identify regions

Table 2. Processor parameters

Parameter	Value
Pipeline width	4
Instruction window/ LSQ sizes	64/32, 128/64, 256/128, 512/256
Branch Predictor	GSHARE with an 8-bit global history, and an 8K BTB
Branch Target Buffer	512 entries, 4-way associativity
Functional units	four integer ALUs, one integer MULT/DIV, two load/stores, four FP adders, one FP MULT/DIV
Latencies	1-cycle integer operations, 3-cycle multiply 2-cycle FP add, 4-cycle multiply 20-cycle integer divide (non-pipelined) 12-cycle FP divide (non-pipelined)
Split L1 I/D caches	64 KB, 2-way set associative, 64 byte lines, 1 cycle hit latency
Unified L2 cache	1 MB, 64 4-way set associative, 64 byte lines, 10 cycle hit latency
DRAM	100-cycle latency, bandwidth of 8 bytes per CPU cycle
HeDGE target fields	2, 3, and 4

of execution that characterize program behavior for a particular input set and simulated these regions. We evaluate seven Java programs from the DaCapo benchmark suite (version `dacapo-2006-10`) [3]. These programs executed 1 billion instructions after forwarding the initialization portion of the execution. In the following discussion, we present results for a subset of programs by including the geometric mean, high and low extremes, and representative samples in each of SPEC INT, SPEC FP, and DaCapo benchmark suites. We refer the reader to a technical report [23] for complete results.

5.2 Performance of HeDGE

This section quantifies the additional `MOV` instructions that HeDGE inserts, their effect on performance, and the contributions due to `MOV` instructions occupying window slots and pipeline bandwidth.

HeDGE introduces `MOV` instructions into the dynamic instruction stream to maintain register dependences when a parent instruction runs out of target entries. These `MOV` instructions behave like regular instructions, and occupy instruction window space and issue and commit bandwidth in the pipeline. Figure 5 plots the percentage of `MOV` instructions that HeDGE adds to communicate dependences for window sizes of 64 to 512 with two, three, four target fields in the reservation stations. We include `MOV`s along misspeculated paths as well, and compute them as a percentage of the total number of committed instructions.

As expected, the number of `MOV`s added decreases as the number of target fields increases. On average, two targets increase executed instructions by 20 to 30%, whereas four targets only increase executed instructions by 2 to 4% (the white portion in the final set of bars). In addition, increasing the instruction window size from 64 to 512 increases the number of `MOV`s. These `MOV` instructions cause a corresponding drop in

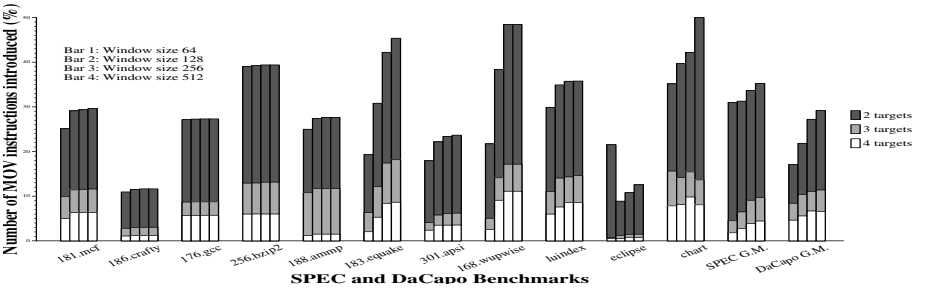


Fig. 5. Percentage HeDGE `MOV` instructions over non-speculative committed instructions in the base processor

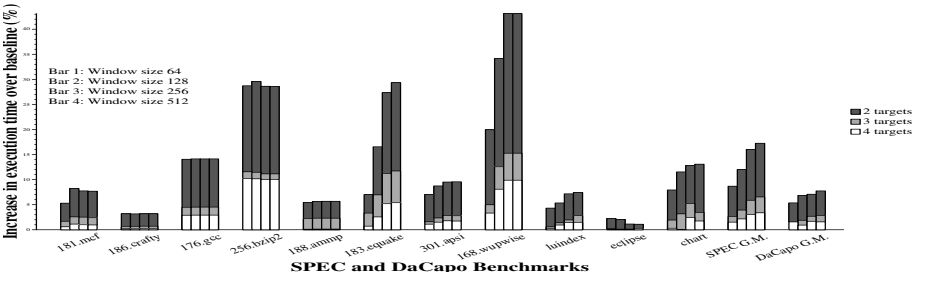


Fig. 6. Increase in execution time for HeDGE over the baseline processor

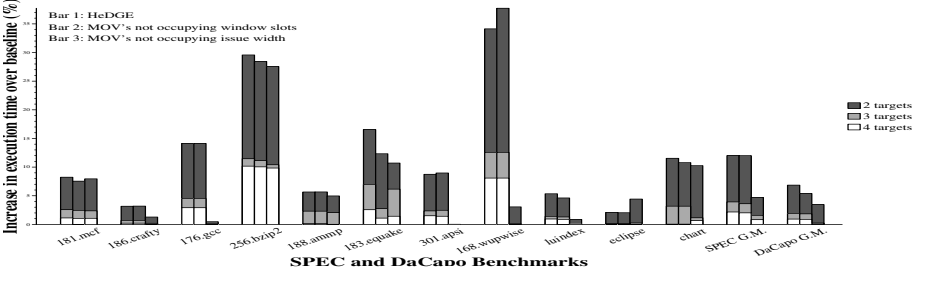


Fig. 7. Instruction window slots and pipeline bandwidth contributions of HeDGE `MOV`s. The first column shows the performance of HeDGE relative to the baseline, the second column shows HeDGE if `MOV`s do not occupy instruction window slots, and the third column shows `MOV`s not consuming pipeline bandwidth.

performance as shown in Figure 6. These plots compare the performance in simulated cycle counts of HeDGE over the baseline configuration. Cycle counts are a more appropriate comparison point than Instructions Per Cycle (IPC) in this work because HeDGE adds additional instructions. For a HeDGE implementation with only two targets, execution time increases by 17%. This number falls to 6.5% for a HeDGE implementation with three targets, and to less than 3.4% for four targets. The increase in execution time for

the DaCapo programs is even smaller, ranging from 8% with two targets, to 1.5% with four target fields. Although these results suggest using four targets rather than two in the reservation stations, four targets require four ports and parallel logic to wakeup four instructions at once. This increased complexity thus favors a lower numbers of targets, but performance favors more targets.

We now further quantify three effects of MOV instructions: (1) They occupy instruction window entries and reducing the effective size of the instruction window. (2) They utilize issue and commit bandwidth, which reduces the effective width of the pipeline. (3) They occupy execution units, conceptually introducing bubbles in the pipeline. To measure the first effect, we assign MOVs to their own window, thereby using the entire instruction window for other instructions. To measure the second effect, we model separate, special issue logic that only executes MOV instructions. Figure 7 plots increases in execution time over a baseline configuration with a moderately aggressive instruction window size of 128, for the SPEC and DaCapo benchmarks. The three columns in the figures show HeDGE, HeDGE when MOV instructions do not occupy instruction window slots, and HeDGE when MOV instructions do not consume pipeline width.

These results show that for both SPEC and DaCapo programs, MOV instructions occupying pipeline bandwidth is the main reason for HeDGE performance degradations; i.e., the third bar in which MOVs do not occupy execution is on average much lower than the other two. A few counterintuitive performance degradations occur when MOV instructions do not occupy instruction window slots, e.g., the second bar is higher than HeDGE for 168.wupwise (and 171.swim and 172.mgrid, not shown in the graphs). When MOV instructions reside in their own buffers, the effective window size for regular instructions increases and there are now more instructions in the window. As a result, HeDGE must add more MOV instructions to the pipeline, which utilize pipeline bandwidth and cause a drop in performance.

To execute a fanout instruction, the processor does not need Arithmetic Logic Units (ALUs) or commit width. Since the only purpose of MOV instructions is to wake up dependent instructions, the issue logic could include an additional bypass path that implements the MOV instructions, waking up dependent consumers.

5.3 Energy Characteristics

We used CACTI 4.2 to model the power and energy characteristics of a conventional CAM-based instruction window and a RAM-based HeDGE design. We use CAM entries with 64 decoded instruction bits and four ports—to support broadcasting up to four physical register tags every cycle. The HeDGE RAMs window adds two to four target fields to every instruction. For a 4-issue processor, the RAM requires four read ports, but eight write ports. Since each instruction has at most two operands, HeDGE needs two write ports to install the target fields in each operand producer for each issued instruction. Given two to four target fields, HeDGE uses eight to sixteen one-bit write ports to set the ready bits of consumer instructions. The number of ports is equal to the number of target fields in the reservation station times the issue width which indicates the maximum number of instructions HeDGE can wakeup in a single cycle.

Table 3 shows the energy consumed per access and leakage power, for 100, 70, and 45 nm technology nodes. CACTI does not currently provide leakage power for 45nm

technology. Although, the HeDGE RAM structure occupies more area than the CAM because it has more ports, the HeDGE RAM consumes 94 to 98% less energy per access than the CAM design. These results show that the CAM leaks 72 to 87% more power than the HeDGE RAM. For the HeDGE RAM, leakage power increases as a function of target fields because each field requires additional transistors.

Table 3. Energy per access (nJ)

IW Size	Energy per access (nJ)				Leakage power (mW)			
	Baseline	HeDGE			Baseline	HeDGE		
		2	3	4		2	3	4
100 nm technology								
64	0.336	0.016	0.014	0.020	14.619	1.965	2.578	2.472
128	0.524	0.019	0.020	0.029	21.703	4.142	4.489	4.530
256	0.932	0.026	0.027	0.030	42.582	7.796	8.501	10.824
512	1.748	0.036	0.038	0.041	84.340	15.823	17.237	21.890
70 nm technology								
64	0.149	0.007	0.007	0.009	60.959	9.339	10.184	11.630
128	0.227	0.008	0.009	0.013	82.736	19.313	21.010	21.727
256	0.403	0.011	0.012	0.017	162.874	37.107	40.540	43.352
512	0.756	0.017	0.018	0.021	323.149	71.437	78.303	89.341
45 nm technology								
64	0.058	0.002	0.003	0.004	Leakage power numbers not available			
128	0.091	0.003	0.003	0.005				
256	0.161	0.004	0.004	0.005				
512	0.303	0.006	0.006	0.007				

We now compare the overall power and energy-delay product of HeDGE against the baseline design. We use energy consumption data obtained from CACTI for the individual structures. We do not include leakage power when computing total energy-delay because how to estimate its contribution to total power is still an open research problem. We rely on prior results for the relative contribution of the instruction window to overall processor power [4,10,11].

Let $e_{Baseline}$ and e_{HeDGE} respectively be the energy consumed each clock cycle by the baseline design and HeDGE. Let $e_{Baseline,scheduler}$ and $e_{HeDGE,scheduler}$ respectively be the energy consumed each clock cycle for each structure. Let f be the contribution of dynamic scheduling towards the overall power consumption in the baseline design. Following Amdahl's law, we obtain $e_{Baseline} = e_{Baseline,scheduler}/f$ and $e_{HeDGE} = e_{HeDGE,scheduler} + (1 - f)e_{Baseline}$. The total energy (E) consumed while executing a program, by the baseline and HeDGE designs, are related by $E_{Baseline}/E_{HeDGE} = (e_{Baseline}.Cycles_{Baseline})/(e_{HeDGE}.Cycles_{HeDGE})$. This ratio is independent of clock frequency, and assumes that the baseline and HeDGE clocks run at the same frequency. We do not take into account that HeDGE structures have a faster access time and hence could be clocked faster.

For 70nm technology, a conservative 10% contribution of the issue logic to total power, and a 512-entry instruction window, Figure 8 plots the relative energy-delay for

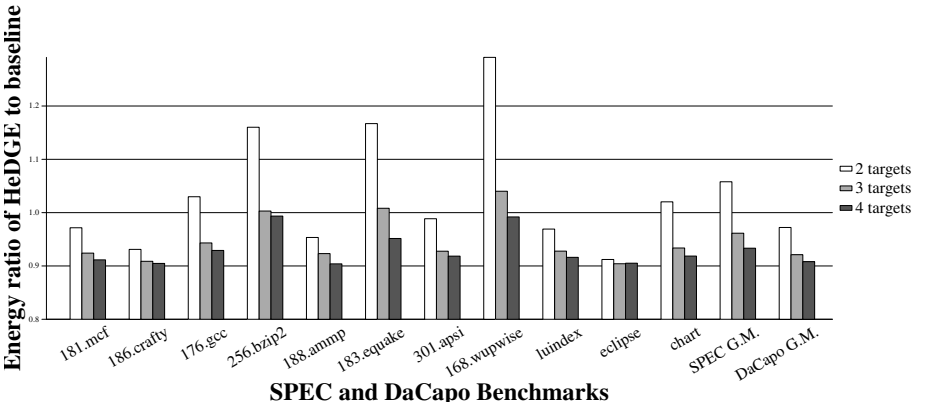


Fig. 8. Energy ratio of HeDGE to the baseline with a 512 instruction window

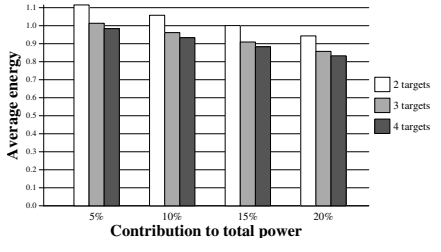


Fig. 9. Energy ratio as a function of the issue logic's total contribution to power

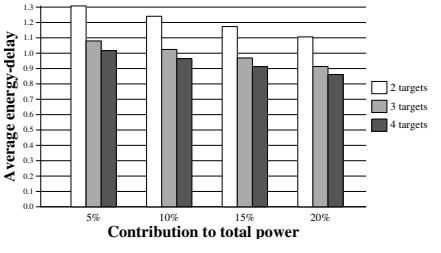


Fig. 10. Energy-delay ratio as a function of the issue logic's total contribution to power

HeDGE designs with two, three, and four target fields compared to the baseline superscalar processor design for all benchmarks. Figures 9 and 10 show the energy and energy-delay as a function of the contribution of the issue logic to total power for ranges from 5 to 20%. Each figure plots the geometric mean of all the benchmarks for 512-entry instruction windows with two, three, and four target fields for 70nm technology. These graphs show that even if a CAM-based processors consumes only 5% of total energy, a four-entry HeDGE RAM improves total energy and energy-delay. If current CAM-based designs are consuming 20% of total power, HeDGE offers significant advantages even with only two target fields.

6 Conclusion

Prior work has shown that the central CAM structure in the issue logic scales poorly with respect to power and latency, and that the issue logic is an integral component of the critical path in superscalar processors. We present HeDGE, a new, more scalable design for the instruction issue logic. HeDGE dynamically transforms instruction dependences implicitly encoded in the register names from a conventional ISA into explicit dependences

by adding target fields to the reservation stations and `MOV` instructions to the instruction stream. HeDGE modifies only the issue, register renaming, and wakeup logic. The main advantage of this design is that it naturally leads to the use of a RAM as the central structure in the issue logic instead of a CAM. We show that even without quantifying the cycle advantages RAMs offer, HeDGE offers substantial power improvements for the issue logic. Furthermore, these results translate to improvements in total processor power, energy, and energy-delay.

References

1. Abella, J., Canal, R., González, A.: Power- and Complexity-Aware Issue Queue Designs. IEEE Micro. 23(5), 50–58 (2003)
2. Alpern, B., Attanasio, D., Barton, J.J., Cocchi, A., Flynn Hummel, S., Lieber, D., Mergen, M., Ngo, T., Shepherd, J., Smith, S.: Implementing Jalapeño in Java. In: ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications, Denver, CO (November 1999)
3. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications, Portland, OR (October 2006)
4. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In: International Symposium on Computer Architecture, Vancouver, British Columbia, Canada, pp. 83–94 (2000)
5. Burger, D., Austin, T.M.: The Simplescalar Tool Set Version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin (June 1997)
6. Canal, R., González, A.: Reducing the complexity of the issue logic. In: International Conference on Supercomputing, Sorrento, Italy, pp. 312–320 (2001)
7. Dennis, J.B., Misunas, D.P.: A Preliminary Architecture for a Basic Data-Flow Processor. In: International Symposium on Computer Architecture, pp. 126–132 (1975)
8. Fields, B., Rubin, S., Bodík, R.: Focusing Processor Policies via Critical-Path Prediction. In: International Symposium on Computer Architecture, Göteborg, Sweden, pp. 74–85 (2001)
9. Folegnani, D., González, A.: Energy-Effective Issue Logic. In: International Symposium on Computer Architecture, Göteborg, Sweden, pp. 230–239 (2001)
10. Gewnnap, L.: Intel's P6 uses Decoupled Superscalar Design. Microprocessor Report 9(2), 9–15 (1995)
11. Gowan, M.K., Biro, L.L., Jackson, D.B.: Power Considerations in the Design of the Alpha 21264 Microprocessor. In: Design Automation Conference, pp. 726–731 (1998)
12. Hamerly, G., Perelman, E., Lau, J., Calder, B.: Simpoint 3.0: Faster and More Flexible Program Phase Analysis. The Journal of Instruction-Level Parallelism 7 (September 2005)
13. Huang, M., Renau, J., Torrellas, J.: Energy-Efficient Hybrid Wakeup Logic. In: ISLPED 2002: Proceedings of the 2002 International Symposium on Low Power Electronics and Design, Monterey, California, USA, pp. 196–201 (2002)
14. Huang, X., Moss, J.E.B., McKinley, K.S., Blackburn, S.M., Burger, D.: Dynamic Simplescalar: Simulating Java Virtual Machines. Technical Report TR-03-03, Department of Computer Sciences, The University of Texas at Austin (February 2003)
15. Kessler, R.E.: The Alpha 21264 Microprocessor. IEEE Micro. 19(2), 24–36 (1999)

16. Lebeck, A.R., Koppanalil, J., Li, T., Patwardhan, J., Rotenberg, E.: A Large, Fast Instruction Window for Tolerating Cache Misses. In: ISCA 2002: Proceedings of the 29th annual International Symposium on Computer Architecture, Anchorage, Alaska, pp. 59–70 (2002)
17. Michaud, P., Seznec, A.: Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors. In: HPCA 2001: Proceedings of the 7th International Symposium on High-Performance Computer Architecture, Monterrey, Mexico (2001)
18. Nagarajan, R., Sankaralingam, K., Burger, D., Keckler, S.W.: A Design Space Evaluation of Grid Processor Architectures. In: MICRO 34: Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture, Austin, Texas, pp. 40–51 (2001)
19. Önder, S., Gupta, R.: Superscalar Execution with Direct Data Forwarding. In: International Conference on Parallel Architectures and Compilation Techniques, pp. 130–135 (1998)
20. Palacharla, S., Jouppi, N.P., Smith, J.E.: Complexity-Effective Superscalar Processors. In: ISCA 1997: Proceedings of the 24th annual International Symposium on Computer Architecture, Denver, Colorado, United States, pp. 206–218 (1997)
21. Sato, T., Nakamura, Y., Arita, I.: Revisiting Direct Tag Search Algorithm on Superscalar Processors. In: Workshop on Complexity-Effective Design (2001)
22. SPEC. Standard Performance Evaluation Committee, `http://www.spec.org`
23. Subramanian, S., McKinley, K.S.: HeDGE: Hybrid Dataflow Graph Execution in the Issue Logic. Technical Report 2008-42, Department of Computer Sciences, The University of Texas at Austin (2008)
24. Swanson, S., Michelson, K., Schwerin, A., Oskin, M.: WaveScalar. In: MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, San Diego, CA, pp. 202–291 (2003)
25. Tarjan, D., Thoziyoor, S., Jouppi, N.P.: CACTI 4.0. Technical Report WRL-2006-86, Hewlett-Packard Labs, Palo Alto (June 2006)
26. Weiss, S., Smith, J.E.: Instruction Issue Logic for Pipelined Supercomputers. SIGARCH Comput. Archit. News 12(3), 110–118 (1984)

Compiler Controlled Speculation for Power Aware ILP Extraction in Dataflow Architectures

Muhammad Umar Farooq, Lizy John, and Margarida F. Jacome

Department of Electrical and Computer Engineering
The University of Texas at Austin
ufarooq@mail.utexas.edu, ljohn@ece.utexas.edu

Abstract. Traditional predicated execution uses two techniques: top predication – in which only the head of the dependence chain is predicated, and bottom predication – in which only the tail of the dependence chain is predicated. Top predication prevents speculative execution, thus delivering minimum performance at minimum energy cost, while bottom predication allows full speculation of the dependence chain, resulting in maximum performance at maximum energy cost. In this paper, we propose a novel *power-aware* ILP extraction technique, denoted the *'elastic-block'*, that combines these two extremes, exposing superior energy vs. performance trade-offs. Each instruction in the elastic-block is explicitly guarded by two predicates: the *speculative*, and the *final*. Instruction's final predicate is generated using traditional if-conversion technique, while the speculative predicate has its default value statically assigned by the compiler, enabling it to make power-performance trade-offs in the code. Several energy saving code optimizations are proposed for the elastic-block structure.

Keywords: Tiled dataflow architectures, predication, power-performance trade-offs.

1 Introduction

The formidable increases in raw transistor density projected for the next 10-15 years pose tremendous scalability challenges to future processor designs, as to how effectively use such devices. Tiled architectures, such as TRIPS, WaveScalar and RAW [1][2][3] exhibit very promising characteristics in that respect –namely, their decentralized organization eliminates several key scalability bottlenecks found in conventional superscalar processors, and reduces overall circuit complexity, effective wire delays and verification effort [4][5]. These favorable characteristics make tiled architectures highly relevant to the future of high performance computing. Large machines, exploiting the huge numbers of raw transistors, possible to integrate in future silicon technologies, can be built in a scalable way, by simply instantiating many such basic tiles on a processor's chip, and then hierarchically organizing them in a suitable way, see e.g. [1][2][3][6]. Aggressive instruction-level parallelism (ILP) extraction is key to the performance of tiled architectures, including WaveScalar, TRIPS, and RAW. Yet, performance/speed alone is not

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 324–338, 2009.

sufficient to quantify the effectiveness of such machines – achieving high energy efficiency is equally critical, so as to aggressively reduce the machine's energy consumption and power dissipation for a given performance point. In this paper, we propose a new *power-aware* ILP extraction technique, denoted the *'elastic-block'*, and show that it exposes superior energy vs. performance trade-offs for tiled architectures. We implemented the elastic-block on the WaveScalar ISA and computing model [2], so as to experimentally demonstrate its effectiveness on a concrete representative of the state-of-the-art in tiled dataflow architectures. Namely, we show that, by using the elastic-block structure, one can deliver almost the same performance of state-of-the-art aggressive ILP extraction techniques, while reducing the average number of instructions executed by 5.95%, and up to 9.95% for some benchmarks, and the average number of messages exchanged between instructions by 6.4%, and up to 23% for some benchmarks – which directly translates in enhanced energy efficiency.

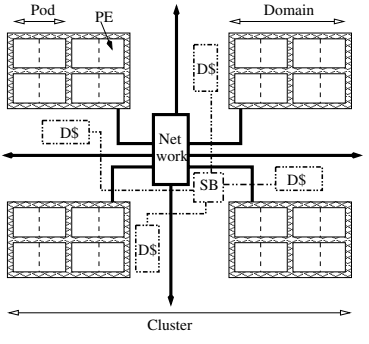


Fig. 1. WaveScalar Microarchitecture

In the next section we will give an overview of our target machine, WaveScalar. Section 3 will introduce elastic-block and its characteristics. Section 4 will explain implementation of elastic-block on WaveScalar. Our evaluation methodology and results are shown in section 5. Section 6 discusses related work in this area. Finally, in Section 7, we disscuss future work and conclude the paper.

2 Overview of Target Tiled Dataflow Machine: WaveScalar

WaveScalar is a dataflow architecture. As in other dataflow architectures, a program is represented as a dataflow graph and instruction dependencies are explicit [7][8]. There is no program counter, instructions are fetched and placed on the grid as they are required. There is no register file, the result produced by an instruction is directly communicated to all the consumers. In this architecture, instructions are grouped in blocks called waves. Waves can be defined as acyclic dataflow graphs for which each instruction executes at most once every time the

wave is executed, and to which control can enter at a single point. On exit and re-entry to this acyclic dataflow graph, the wave-number is increased.

Each dynamic instruction is identified by a tag which is the aggregate of its wave-number and location on the grid. When an instruction has received all its input operands for a particular matching wave-number, it fires, provided there is room to store the result in the output queue, and an ALU is available. The output is temporary stored in the output queue before it is communicated to the consumers. Figure 1 shows the basic WaveScalar Microarchitecture. The substrate consists of replicated clusters connected through a dynamically routed packet network. Each cluster consists of four domains, communicating through a fixed-route network switch which has a 4 cycle latency. Additionally, each cluster has a 32KB 4-way set associative L1 data cache, and a store buffer. Each domain is composed of eight processing elements (PEs), grouped into pairs of two. Each pair is called a pod. Pods communicate through a fixed 1 cycle latency pipeline network. Within PE instructions communicate through a bypass network. Figure 2 shows the 5-stage in-order PE pipeline. Each PE has a small instruction cache capable of holding 64 static instructions. Each PE has a 16 entry input queue and an 8 entry output queue.

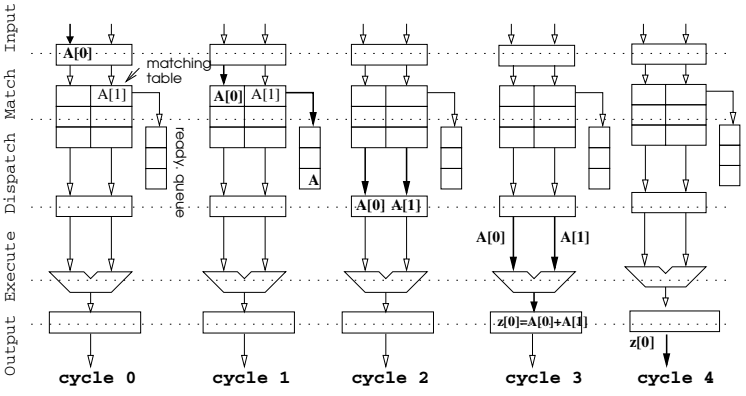


Fig. 2. Processing Element (PE) pipeline stages

2.1 Pipeline Stages of a PE

1. **INPUT:** Accepts input operand messages arriving from other PEs and from itself, and places the operands in the pipeline registers for the next stage.
2. **MATCH:** Operands are moved from the pipeline registers to the matching table at an index computed by XOR hash of the wave-number, thread-id, and destination instruction number of each operand. MATCH also determines which instructions have all their operands with the matching wave-number, thread-id and are ready to fire. It then issues all ready instructions to the DISPATCH stage by placing their matching table index in the ready queue.

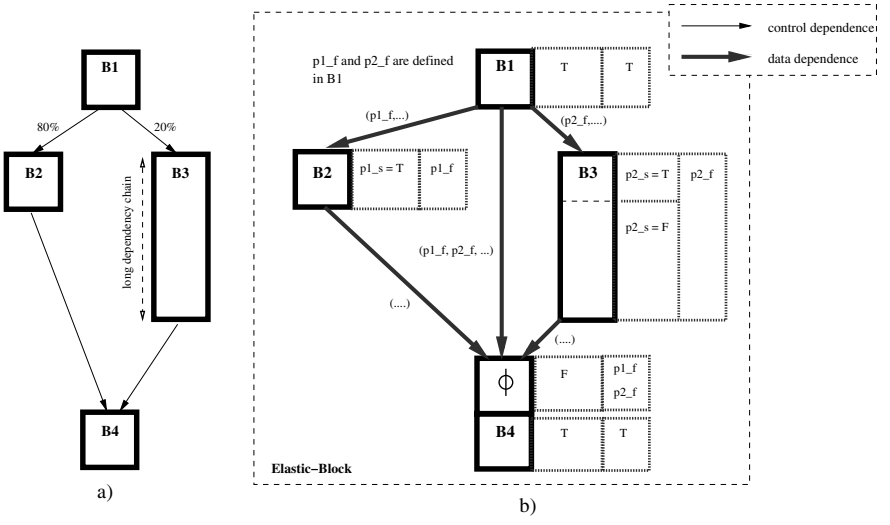


Fig. 3. Illustrating elastic-blocks. (a) Original control flow graph annotated with the dynamic frequency of execution of basic-blocks B2 and B3. B3 is depicted with a longer box relative to B2 in order to represent the fact that it contains much longer dependence chains. (b) Structure of resulting elastic-block code, with distinct default values assigned to the speculative predicate of instructions within B3, depending on their depth in the corresponding dependence chains.

3. **DISPATCH:** Removes the matching table indices from the ready queue, reads the corresponding operands from the matching table and forwards them to EXECUTE stage for execution.
4. **EXECUTE:** Instruction executes sending its results to the output queue.
5. **OUTPUT:** Removes the entries from the output queue sending them to the consumer instructions.

For better understanding of WaveScalar microarchitecture, we refer the readers to [9].

3 Power Aware ILP Extraction with Elastic-Blocks

In this section, we introduce the elastic-block and discuss its operation and power-aware features, in contrast to previous techniques.

3.1 The Elastic-Block Structure: Definition, Power-Aware ILP Extraction, and Energy Saving Code Optimizations

Guarding instructions with speculative and final predicates. The key innovation introduced in the elastic-block structure is the ability to explicitly guard instructions with two predicates – the *speculative*, and the *final* predicate.

The instruction's *final* predicate is generated using traditional if-conversion technique [10][11]. Specifically, all control dependencies within the code region targeted for elastic-block formation are converted into data dependencies, similarly to what is done, e.g., in hyperblock [12], conditional branches are replaced with comparison instructions which set the *final* predicate of all instructions that are control dependent on such branches. Even if generated using well known techniques, the final predicate is a very unique operand type in our target dataflow ISA, in that an instruction may actually execute even if the value of its final predicate is still unknown. Speculative predicates, in turn, always have a default value statically assigned by the compiler – as it will be seen, different such assignments can implement distinct power-performance trade-offs in the code. Speculative predicates explicitly enable *control speculation* – that is, when the speculative predicate of an instruction is set to TRUE, if all 'regular' operands of that instruction become available, while the value of its final predicate is still *unknown*, the instruction becomes ready for execution.

We noted that within an elastic-block, the compiler can *selectively* and *individually* define which instructions should be speculatively executed, and which should not. Indeed, while the value of the final predicate of all instructions within a basic-block must necessarily be identical, this need not be the case for their corresponding speculative predicates – the fact that each instruction keeps its own copy of the speculative predicate in our target tiled dataflow architecture allows such a discrimination to be made in a very natural/simple way. Of course, the value of the speculative predicate, as the name suggests, is only relevant while the instruction's final predicate is not available – namely, if an instruction receives its final predicate while still waiting for other (regular) operands, the value of the final predicate alone determines if the instruction will be executed or squashed prior to execution.

In turn, if the speculative predicate of an instruction is set to FALSE, then control speculation is explicitly disabled, that is, the instruction will not be ready for execution until it actually receives its final predicate value. If such final predicate happens to be FALSE, the instruction is locally squashed *prior* to execution. Otherwise, it, of course, executes. Note that, the semantics of our final predicate is, thus, somewhat different form that of the standard predication model adopted, for example in [13], due to performance reasons, instructions always execute, and their predicates are only used to decide if their results should be committed or not. Reflecting that fact, traditional hyperblock selection approaches, such as [12], do not favour the inclusion of large basic-blocks in a hyperblock, since such blocks utilize many machine resources and may actually end up negatively impacting performance, as opposed to enhancing it. As will be seen below, the elastic-block's increased flexibility enables such aggressive performance-enhancing ILP extraction techniques to be enhanced with energy awareness and efficiency considerations.

Note finally that, whenever control speculation is explicitly enabled in the elastic-block, ϕ functions may need to be inserted in the corresponding code, so as to potentially reconcile multiple (speculative) definitions/updates of the

same variable, i.e., make sure that only the 'right' value is actually sent to the corresponding consumers. As discussed in more detail in Section 4, such ϕ functions are implemented by *move* instructions, each guarded by the same final predicate used on the actual basic-block where the value, being sent, was generated.[1]

Simple illustrative example of power-aware ILP extraction using the elastic-block. Consider the weighted control graph shown in Figure 3(a). In this simple example, a conditional branch instruction in basic-block B1 defines two control paths, one through basic-block B2 and another through basic-block B3. Although the control path through B3 is taken much less frequently than that through B2 (on average 1 out of 5 times, as indicated in the figure), B3 also contains much longer data dependence chains. So, even if executed infrequently, B3 takes much longer than B2 to complete, so much so that it does actually impact overall performance. Assume also, that control speculation would substantially improve performance for this code segment, i.e., performance can be enhanced by starting to execute B2 and/or B3's instructions, prior to knowing which control path will be taken. Figure 3(b) illustrates how such performance can be delivered, in an energy efficient way, using the elastic-block structure.

Note first that, as alluded to before, all control dependencies in the elastic-block region have been converted into data dependencies using standard if-conversion – as indicated in Figure 3(b), final predicates (denoted as p1_f and p2_f) guard the instructions originally in basic-blocks B2 and B3 respectively, and their corresponding predicate-define instructions have been 'inserted in B1'. Note further that, in the simple example of Figure 3, B1 and B4 represent simple straight line code that always executes, and thus, the compiler can directly assign the default value TRUE to the speculative predicate of the corresponding instructions, thereby, eliminating the need for final predicate.

The key idea in energy-aware ILP extraction is to enable the selective speculation of *only* those instructions that may actually payoff in terms of performance enhancement, thus avoiding wasteful energy spending. In the case of the illustrative code segment shown in Figure 3, for example, the compiler has detected that a performance gain can be achieved by speculating *all* of the instructions in the most commonly executed block, i.e., B2, and thus it did set the default value of their corresponding speculative predicates (denoted p1_s in Figure 3(b)) to TRUE. Accordingly, the instructions in B2 will become ready for execution as soon as they receive all of their 'regular' operands, but their final predicate. In addition, the compiler has detected that speculatively executing a *select subset* of the instructions in B3, namely, those located 'early' in B3's long dependence chains, would also give a relevant performance gain. Accordingly, it did set the speculative predicates of that select subset of B3's instructions to TRUE, and assigned FALSE to the remaining (see p2_s values in Figure 3(b)). As alluded to before, since each instruction keeps its own copy of the speculative predicate, such a discrimination can be made in a very simple way. Note finally that, as

[1] Naturally, the speculative predicate of such *move* instruction is always FALSE, see Figure 3(b).

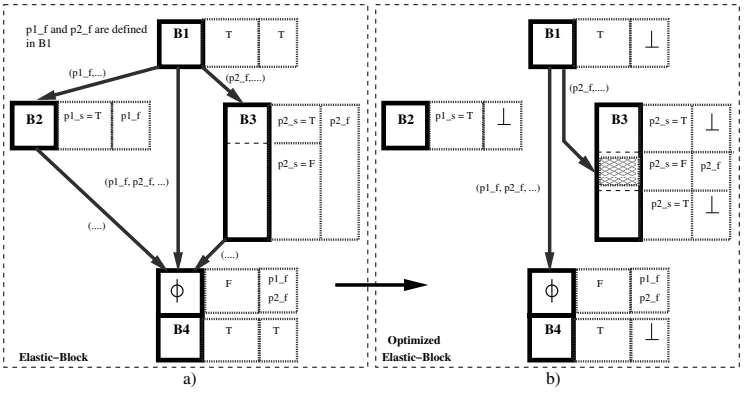


Fig. 4. Illustrating elastic-block code optimizations. (a) Non-optimized version. (b) Optimized version with elimination of final predicate messages. The edges in figure (b) represent only final predicate messages, for clarity. As it can be seen, the final predicate is no longer sent to the instructions originally in B2 – this is represented by placing the symbol bottom ($\perp$) in the corresponding field. In fact, the final predicate is now only sent to the first non-speculative instruction in the dependency chains of B3. Specifically, instructions that have data dependencies to these need not receive their final predicate as well, and in fact can be again made speculative, since they cannot execute unless their non-speculative predecessors send them their operands. So, we use transitivity effects to eliminate again final predicate messages.

indicated above, although the final predicates guarding B2 and B3's instructions are necessarily mutually exclusive (i.e., p2_f != p1_f), the speculative predicates guarding these blocks need not be, and in fact frequently will not be – this is why we have adopted naming conventions explicitly distinguishing among such predicates.

Energy Saving Code Optimizations. When an instruction is speculatively executed, yet its final predicate turns out to be FALSE, there is nothing to be done in the local context of the instruction – as alluded to above, the ϕ functions in the elastic-block code will make sure that only correct values are actually sent to the appropriate consumers. In fact, if the compiler can determine that the final predicate of an instruction will never arrive prior to its speculative execution, or will rarely do so, the message containing the final predicate should not even be sent to that particular instruction, thus reducing energy consumption as well as message traffic – this is one of the key energy saving optimizations that can be performed in elastic-block code, symbolically illustrated in block B2 and in the upper third of block B3, by placing the bottom or 'absence' symbol ($\perp$) in the corresponding final predicate fields, see Figure 4(b).

A second type of energy saving optimization can be done by directly relying on the very nature of the dataflow model and exploiting the transitivity of data dependencies inside a basic-block – this second type of optimization is symbolically illustrated in the bottom third of block B3 of Figure 4(b). Specifically,

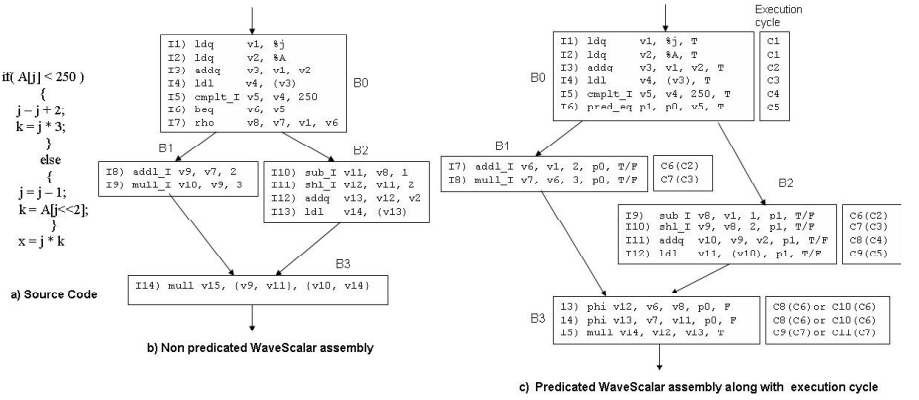


Fig. 5. Example of if-then-else predication/speculation. (a) source code, (b) non-predicated WaveScalar assembly code segment, (c) assembly code segment after predication along with default setting for speculative predicate. General format for the predicated WaveScalar assembly instruction is 'opcode destination(s), source(s), final predicate, speculative predicate'. Note that since basic-block B0 is a straight line code, its speculative predicate is set to T, eliminating the need for final predicate. In B1 and B2, depending on power-performance trade-off, amount of speculation can be adjusted from any where between full speculation to no speculation by setting the default value for speculative predicate appropriately.

any non-speculative instruction that consumes data from at least one other non-speculative instructions that is guarded by the same final predicate, can be immediately converted into a speculative instruction, since it cannot possibly execute unless the producer of that operand has already executed – so, by taking advantage of such transitivity within a basic-block, no final predicate needs to be explicitly sent to these 'dependent instructions', as indicated by placing the bottom symbol ($\perp$) in the corresponding field in Figure 4(b).

Implementation details on the WaveScalar architectures will be given in Section 4, and the impact of the above optimizations will be experimentally quantified for representative benchmarks, in Section 5.

4 Implementation on WaveScalar

This section will explain our implementation of elastic-block, and related optimizations, on a concrete representative of the state-of-the-art in tiled dataflow architectures, WaveScalar.

4.1 ISA Extensions

Figure 5(a) and 5(b) shows a simple if-then-else construct and its corresponding non-predicated WaveScalar assembly code. Figrue 5(c) shows our modified predicated WaveScalar assembly code. Changes are explained as below:

Adding speculative and final predicates: In the modified code, the execution of each instruction is guarded by two additional 1-bit operands – *final* predicate and the *speculative* predicate. Final predicate receives its value from *predicate-define* instruction (I6 in Figure 5(c)), which is also part of our ISA extension. Unlike final predicate, the value of speculative predicate is set by the compiler (either T or F) through program analysis, thus enabling the compiler to make power-performance trade-off in the code. Note that, basic-block B0 contains straight line code, its speculative predicate is set to T, eliminating the need for final predicate. Basic-blocks B1 and B2 have p0 and p1 respectively as their final predicate.

Addition of phi (ϕ) instruction: In Figure 5(c), instructions from basic-blocks B1 and B2 can execute speculatively (if speculative predicate is set to T). This requires addition of ϕ instructions (I13, I14 in Figure 5(c)) in the merge block B3. ϕ instruction takes two input values and a final predicate and, depending on the final predicate value, produce one of the inputs on its output. For correct execution, ϕ instruction can't execute speculatively and should wait for final predicate to arrive.

Removing rho (ρ) instruction: Figure 5(b) shows non-predicated, non-speculative WaveScalar assembly code. Instructions are executed only from the *'taken'* path. Instructions from *'not-taken'* path are prevented from execution by blocking their input operands using rho (ρ) instruction. The *rho (ρ)* instruction (I7 in Figure 5(b)), is a conditional split instruction. The ρ instruction takes an input value and a boolean output selector. It directs the input to one of two possible outputs depending on the selector value, effectively steering data values to the instructions in either basic-block B1 or B2. Speculative execution, however, allows execution from both basic-blocks i.e. B1 and B2. This is achieved by removing the ρ instruction and directly connecting its input operand with the input operands of its destination instructions.

4.2 Microarchitecture Support for Predicated and Speculative Execution

This section will explain microarchitecture modifications to the PE pipeline stages in order to support predication, and speculation.

Processing Element Modifications. Changes were made in the first two stages, namely INPUT and MATCH, of the PE pipeline described in Section 2.

1. Modified INPUT stage: Accepts input operands arriving from other PEs and from itself with the following additional logic: If the arriving operand is a '*final predicate*' operand with a FALSE value, the corresponding instruction is squashed by invalidating its entry in the matching table. However, this can lead to two special situations. Firstly, late arriving operands of an already squashed instruction will get a permanent entry in the matching table. Secondly, consumers of an squashed instruction keep waiting for the operand to arrive. To address the first situation each instruction has its 'current valid wave-number'

Table 1. Evaluating readiness of an instruction

Data operand	Final predicate	Spec. predicate	Action Taken
?	?	*	wait for data to arrive
?	TRUE	*	wait for data to arrive
?	FALSE	*	squash the instruction
data available	?	FALSE	wait for final predicate to arrive
data available	?	TRUE	execute instruction speculatively
data available	TRUE	FALSE	execute instructions normally
data available	FALSE	FALSE	squash the instruction

? = has not arrived, * = don't care

stored in the instruction cache. When an instruction is dispatched to the ready queue or squashed (if its final predicate is FALSE), its wave-number is stored as the 'current valid wave-number'. If the wave-number of an arriving operand is less or equal to 'current valid wave-number', it is not entered in the matching table. Second situation actually can never arise. If all the consumers of an squashed instruction and the squashed instruction itself are in the same basic-block, say B, they all will receive the same final predicate and eventually will be squashed. If however, consumers of an squashed instruction are in the merge block, they will receive their operands from the basic-block whose final predicate evaluates to TRUE i.e. sibling of B.

2. Modified MATCH stage: With non-speculative execution, an instruction only becomes ready to execute when all its operands have arrived. However, speculative execution requires modifying the logic that determines the readiness of an instruction. Table 1 lists all possible cases and the corresponding action taken.

4.3 Power-Performance Trade-Off Using Compiler Analysis

Traditionally the focus of compiler optimization has been on improving performance (see for example [14] and the references therein). However, performance alone is not sufficient to measure the effectiveness of machines. Other metrics such as energy efficiency and power dissipation are equally important. Unfortunately, there has been little effort to analyze the role of compiler in achieving high energy efficiency.

Elastic-block enables the compiler to make power-performance trade-offs in the code. Compared to 'hyperblock' [12], 'elastic-block' is capable of achieving more power-performance trade-off points. During hyperblock formation a basic-block is either fully included or fully excluded. With elastic-block, the compiler can *selectively* and *individually* define which instructions in the basic-block should be speculatively executed, and which should not. During the elastic-block formation, compiler profiles the execution frequeucy of individual basic-blocks, and partitions the instructions into 'levels' based on their dependence depth. Instructions that receive their operands from outside the elastic-block are at level-1. Instructions dependent on level-1 instructions are at level-2 and so on.

Table 2. Power-Performance Trade-off Points obtained using Elastic Block

Amount of Speculation	Cycles Taken	Instructions Executed
No speculation	9*60 + 11*40 = **980**	6*100 + 2*60 + 4*40 + 3*100 = **1180**
Only B1	7*60 + 11*40 = **860**	6*100 + 2*100 + 4*40 + 3*100 = **1260**
B1 + B2	7*60 + 7*40 = **700**	6*100 + 2*100 + 4*100 + 3*100 = **1500**
B1+ {I9} in B2	7*60 + 10*40 = **820**	6*100 + 2*100 + 100 + 3*40 + 3*100 = **1320**
B1 + {I9, I10} in B2	7*60 + 9*40 = **780**	6*100 + 2*100 + 2*100 + 2*40 + 3*100 = **1380**
B1 + {I9, I10, I11} in B2	7*60 + 8*40 = **740**	6*100 + 2*100 + 3*100 + 40 + 3*100 = **1440**

Note: Hyper blocks can only achieve first three points

Based on the execution frequency of the basic-block and the dependence level of the instruction, compiler decides whether to set the speculative predicate of that instruciton to TRUE or FALSE for a given power-performace point. Consider the same example shown in Figure 5(a) and its corresponding assembly in Figure 5(c). Assume that this code executes 100 times with basic-block B1 executing 60 times and basic-block B2 executing 40 times. Basic block B0 contains straight line code, its speculative predicate is set to TRUE, eliminating the need for final predicate. B1 and B2 are guarded by final predicate p0 and p1 respectively. Speculative predicate for instructions in B1 and B2 can be assigned either TRUE or FALSE. For this example, each instruciton is assumed to be single cycle. Instruction's execution cycle (both when executed speculatively and non-speculatively) is also shown in Figure 5(c). Table 2 shows various operating points that are achieved by selective speculation of instructions in a elastic-block structure. First row in Table 2 shows a low performance but most power efficient operating point where no instruction is speculatively executed. Third row shows the best performance but least power efficient point where both basic-blocks B1 and B2 are fully speculated. Rest of the rows shows several operating points between these two extremes.

5 Performance Analysis

5.1 Experimental Methodology

Elastic-block technique and related optimizations are implemented in WaveScalar compiler/binary-translator, and necessary microarchitectural support is provided in the WaveScalar simulator. Speculative execution is supported on all instructions but stores, phi and predicate-define instructions. Benchmarks from SPEC 2000, MediaBench, EEMBC benchmark suites are used for the evaluation. Our experimental setup was designed to evaluate the effectiveness and flexibility of elastic-blocks at exploiting the power-performance trade-off. Several configurations, each with varying depth of speculation, are computed by the compiler, by choosing different values for speculative predicate. For each configuration, the benchmarks are run till completion. IPC is not a meaningful metric in our case, since higher IPC does not necessarily mean higher performance because of 'unnecessary instructions' executed due to predication/speculation. So, we will measure: (1) number of cycles

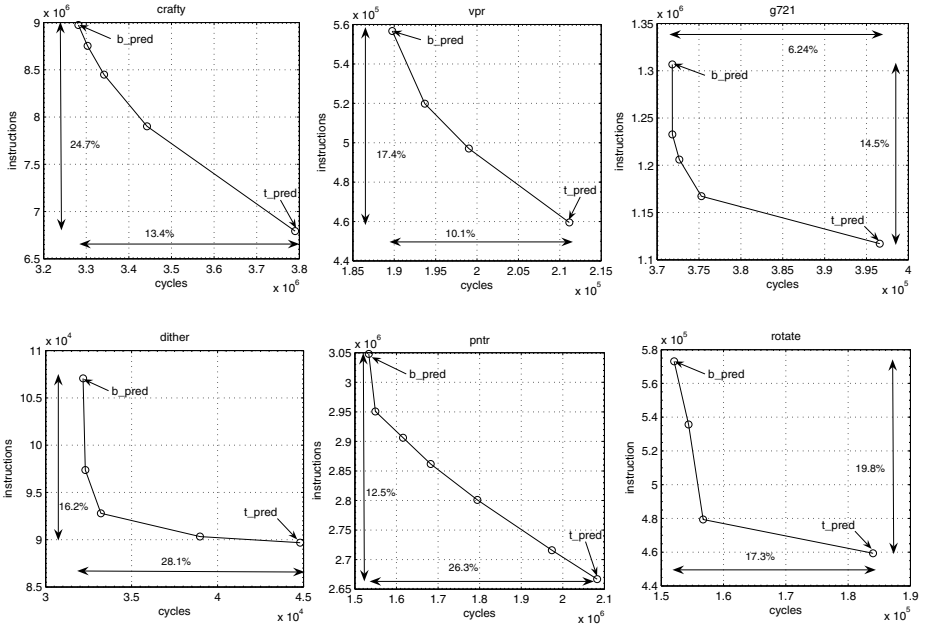


Fig. 6. Power-Performance trade-off points between top predication (t_pred) and bottom predication (b_pred)

required to execute the application, which roughly relates to performance; and (2) corresponding total number of executed instructions; and (3) number of operand and predicate messages exchanged during that execution, which along with the instructions executed corresponds to the power consumed during execution.

5.2 Results

Adding predication and speculation improved WaveScalar average performance by 16.9% (for bottom predication), compared to no speculation (top predication), see Figure 6. However, this increase in performance comes at an steep cost of 17.51% extra instructions executed, and 14.35% additional messages sent, which is unwarrented for high performance, low power computing. Figure 6 and 7 shows that almost similar performance gain, 15.96%, can be achieved with an average 11.56% increase in instructions and 7.95% increase in operand messages, a reduction of 5.95% and 6.4% respectively. Another high performance point with 13.93% performance gain, can be achieved with an average 7.74% increase in instructions and 3.1% increase in operand messages, a reduction of 9.77% and 11.25% respectively. Operand messages, shown in Figure 7, scales with the number of instructions executed. Using the optimization explained earlier in Figure 4(b), predicate messages are independent of the number of instructions executed, see Figure 7.

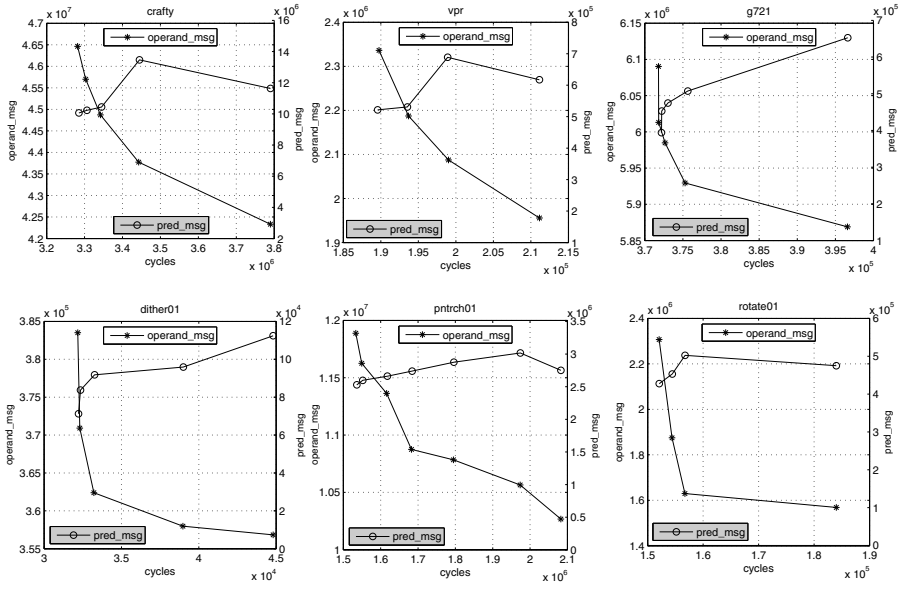


Fig. 7. Illustrating operand and predicate messages for the corresponding power-performance trade-off points shown in Figure 6

6 Related Work

DataFlow Predication for EDGE Architectures
Smith et al. has proposed dataflow predication for EDGE architectures [15]. Smith et al. uses top or bottom predication, in which either the first or the last instruction of a dependence chain is predicated. Top predication delivers low performance and low energy computation, as instructions are not executed speculatively, while bottom predication results in high performance and high energy computation as all the instructions in the dependence chain, except the bottom instruction, are fired speculatively. The focus of this work is to combine these two extremes, allowing the compiler to decide the optimal depth of speculation, for a given power-performance point. Second, in EDGE architecture, each instruction has a two-bit predicate field that specifies whether that instruction is predicated on a TRUE predicate, a FALSE predicate, or unpredicated. However, in our proposed work the speculative predicate is another operand, initially set by the compiler, but later can be modified through messages, thus allowing run-time adaptation, which is a subject of future work.

Predication for Superscalar Architectures
Mahlke et al. proposed a compiler structure, hyperblock, that groups together most frequently executed basic-blocks from different control paths, allowing effective scheduling for these basic-blocks [12]. In case of an hard-to-predict branch (say 60/40), basic-blocks from *'both'* control-flow paths are included in the

hyperblock , and all instructions in these basic-blocks are executed all the time. In our proposed *'elastic-block'* structure, basic-blocks from *'both'* control-flow paths will be included, but speculative execution of instructions in these basic-blocks will be proportional to their execution frequency. Kim et al. combined the use of conditional branches, for easy-to-predict branches, with predicated execution, for hard-to-predict branches [16]. Their motivation for not converting every conditional branch into predicated code is twofold: First, the processor needs to fetch useless instruction, thus wasting the fetch bandwidth. Second, compared to branch prediction in which instructions are executed before the branch is resolved, predicated instructions add extra delay, as they have to wait for the predicate value to be ready. In our proposed work, we transformed all branches to predicated code, as we don't have the aforementioned overheads: First, instructions are stored on the execution grid, once they are fetched from the memory, and second, predicated instructions can execute speculatively before the predicate value is ready (by setting p_s = TRUE).

7 Conclusion and Future Work

A novel *power-aware* ILP extraction technique, that combines predication with speculation, is introduced for tiled dataflow architectures. Each instruction in this flexible structure, denoted the *elastic-block*, is guarded explicitly by two predicate operands: the final predicate, and the speculative predicate. By assigning the default value of speculative predicate to TRUE, compiler can *selectively* and *individually* enable the speculation of *only* those instructions that may actually payoff in terms of performance improvement, thus avoiding wasteful energy spending. This is in contrast to the existing techniques for predicated execution, namely top predication and bottom predication, in which either the head or the tail of the dependence chain is predicated. Results showed that by merging top and bottom predication, and allowing the compiler to determine the depth of speculation, performance close to traditional predication can be delivered while improving the energy efficiency. The key advantage of elastic-block structure will be its inherent potential for run-time adaptivity, and is a subject of future work.

References

1. Burger, D., Keckler, S.W., McKinley, K.S., Dahlin, M., John, L.K., Lin, C., Moore, C.R., Burrill, J., McDonald, R.G., Yoder, W.: The TRIPS Team: Scaling to the End of Silicon with EDGE Architectures. Computer 37(7), 44–55 (2004)
2. Swanson, S., Michelson, K., Schwerin, A., Oskin, M.: WaveScalar. In: MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, p. 291. IEEE Computer Society, Los Alamitos (2003)
3. Waingold, E., Taylor, M., Sarkar, V., Lee, V., Lee, W., Kim, J., Frank, M., Finch, P., Devabhaktumi, S., Barua, R., Babb, J., Amarsinghe, S., Agarwal, A.: Baring It All to Software: The Raw Machine. Technical report, Cambridge, MA, USA (1997)

4. Hrishikesh, M.S., Keckler, S.W., Burger, D., Agarwal, V.: Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. ISCA 00, 248 (2000)
5. Hunt, W.: Introduction: Special Issue on Microprocessor Verification. Formal Methods in System Design, 135–137 (2002)
6. Mai, K., Paaske, T., Jayasena, N., Ho, R., Dally, W.J., Horowitz, M.: Smart Memories: A Modular Reconfigurable Architecture. In: ISCA 2000: Proceedings of the 27th Annual International Symposium on Computer Architecture, pp. 161–171. ACM Press, New York (2000)
7. Dennis, J.B., Misunas, D.P.: A Preliminary Architecture For a Basic Data-flow Processor. SIGARCH Comput. Archit. News 3(4), 126–132 (1974)
8. Papadopoulos, G.M., Culler, D.E.: Monsoon: An Explicit Token-Store Architecture. In: ISCA 1998: 25 years of the International Symposia on Computer Architecture (selected papers), pp. 398–407. ACM Press, New York (1998)
9. Putnam, A., Swanson, S., Mercaldi, M., Petersen, K.M.A., Schwerin, A., Oskin, M., Eggers, S.: The Microarchitecture of a Pipelined WaveScalar Processor: An RTL-based Study. Technical report, Washington, DC, USA (2004)
10. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.: Conversion of Control Dependence to Data Dependence. In: POPL 1983: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 177–189. ACM Press, New York (1983)
11. Park, J.C.H., Schlansker, M.S.: On Predicated Execution. Technical report, Palo Alto, CA, USA (May 1991)
12. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective Compiler Support for Predicated Execution Using the Hyperblock. In: 25th Annual International Symposium on Microarchitecture (1992)
13. Chang, P.P., Mahlke, S.A., Chen, W.Y., Warter, N.J., Hwu, W.m.W.: IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In: ISCA 1991: Proceedings of the 18th Annual International Symposium on Computer Architecture, pp. 266–275. ACM Press, New York (1991)
14. Wolfe, M.: High Performance Compilers for Parallel Computing. Pearson Education POD (1995)
15. Smith, A., Nagarajan, R., Sankaralingam, K., McDonald, R., Burger, D., Keckler, S.W., McKinley, K.S.: Dataflow Predication. In: MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, pp. 89–102. IEEE Computer Society, Los Alamitos (2006)
16. Kim, H., Mutlu, O., Stark, J., Patt, Y.N.: Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution. In: MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, pp. 43–54. IEEE Computer Society, Los Alamitos (2005)

Revisiting Cache Block Superloading

Matthew A. Watkins[1], Sally A. McKee[2], and Lambert Schaelicke[3]

[1] School of Electrical and Computer Engineering
Cornell University
mwatkins@csl.cornell.edu
[2] Department of Computer Science and Engineering
Chalmers University of Technology
sallyamckee@gmail.com
[3] Fort Collins Design Center
Intel Corporation
l.schaelicke@computer.org

Abstract. Technological advances and increasingly complex and dynamic application behavior argue for revisiting mechanisms that adapt logical cache block size to application characteristics. This approach to bridging the processor/memory performance gap has been studied before, but mostly via trace-driven simulation, looking only at L1 caches. Given changes in hardware/software technology, we revisit the general approach: we propose a transparent, phase-adaptive, low-complexity mechanism for L2 superloading and evaluate it on a full-system simulator for 23 SPEC CPU2000 codes. Targeting L2 benefits instruction and data fetches. We investigate cache blocks of 32-512B, confirming that no fixed size performs well for all applications: differences range from 5-49% between best and worst fixed block sizes. Our scheme obtains performance similar to the per application best static block size. In a few cases, we minimally decrease performance compared to the best static size, but best size varies per application, and rarely matches real hardware. We generally improve performance over best static choices by up to 10%. Phase adaptability particularly benefits multiprogrammed workloads with conflicting locality characteristics, yielding performance gains of 5-20%. Our approach also outperforms next-line and delta prefetching.

1 Introduction

A program's memory performance is determined both by working set sizes and by temporal and spatial locality within working sets. Improving temporal locality requires converting long-distance data reuse to short-distance cache reuse by regrouping computations sharing data. Such optimizations are usually implemented in software by compiler optimizations, such as loop interchange, blocking [27], fusion, and more complex regrouping [11,15,17,18], or programmer intervention. Spatial locality optimizations like cache-conscious data placement [1] or structure placement [3,5] improve memory performance via software changes,

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 339–354, 2009.

but these require profiling and have not yet become common in compiler toolchains. Such approaches attempt to find good data placements for entire executions, without adapting to phase changes. A related alternative is copying data [13,22,26] at strategic execution points.

Spatial locality optimizations are usually performed in hardware, where straightforward approaches may have high payoff (evidenced by the ubiquity of dynamic caching). Obviously, larger cache blocks exploit spatial locality, but tradeoffs exist between block size, bus utilization, and memory hierarchy efficiency. If most block data are not used, larger blocks (as in superblocks [9] or superloading [25]) may actually *decrease* application performance by increasing cache conflicts and bus, memory controller, and DRAM occupancy.

We evaluate an efficient mechanism to balance costs and benefits of accessing data at larger granularities. Since a single fetch granularity is suboptimal for all applications [8,24], or even for different phases of a given application's execution, our mechanism must adapt dynamically. Our approach minimizes additional cache complexity, has no impact on processor cycle time, allows multiple logical block sizes, permits non-unit (in terms of base cache block size) increases and decreases, and adapts naturally to application phase behavior without application software involvement. Most previous cache block superloading studies are trace-based, use smaller applications, do not model an OS, and focus only on level-1 caches (see Sect. 2). With the rapid pace of technological innovation, "good science" dictates that we reevaluate conclusions from older research. As McKinley and Temam demonstrate [14], common beliefs often fail to hold in practice, and thus reevaluating older results within newer systems has merit.

2 Related Work

Adapting prefetch distance or block size has been studied in several contexts. Dahlgren et al. [4] show that adaptive sequential prefetch distances in hardware improves performance in SMPs, and Gornish and Veidenbaum [7] augment the hardware approach with compiler assists. Space limitations prevent thorough treatment of the rich related work via software, hardware, or hybrid approaches. We thus focus on transparent hardware implementations requiring no ISA changes. For such studies, machine parameters, memory bandwidths, latencies, cache block sizes, simulation methods, and benchmark selection significantly influence results.

The MIPS 3000 cache has configurable line[1] sizes, but only on hardware reset [10]. Early designs accommodating variable line sizes include Seznec's Decoupled Sector Cache [21]. Temam and Jegou [23] fetch adjacent blocks into a separate buffer to exploit spatial locality and avoid cache pollution, and González et al. [6] study dynamically allocating data across caches with different line sizes, one optimizing temporal locality and the other spatial locality.

[1] We use "line" and "block" interchangeably, according to how other authors describe their work.

Kumar and Wilkerson [12] study structured spatial prediction/prefetching for L1 data caches; their online scheme predicts line patterns to be loaded to reduce pollution from superblocks. Chen et al. [2] adapt Spatial Pattern Prediction to larger locality regions via prediction tables exploiting instruction addresses combined with offsets within cache blocks. They target L1s, allowing subblock utilization to save 41% leakage energy, on average, improving performance up to 2× while maintaining performance within 1% on 12 SPEC CPU2000. They arbitrarily track simulation statistics 10 billion instructions into each execution; and data represent only 500 million instructions; furthermore, they do not model OS activity. Our findings contradict theirs: in many cases we attribute this to our studying whole program behaviors. Direct comparisons are difficult, given differences in simulation methods and memory hierarchy levels targeted.

Van Vleet et al. [25] use trace-driven simulation of nine SPEC CPU95 benchmarks to study effects of online superloading decisions for an L1 cache. They track usage among adjacent lines to decide whether to perform a superload. A separate Line Size Detector state machine later determines what the best size would have been for a given load. Their goal is to study superloading potential, and thus they avoid aliasing by using unlimited space. They compare dynamic performance to optimal sequences of loads/superloads determined offline, finding their hardware mechanisms perform competitively.

Veidenbaum et al. [24] use trace- and execution-driven L1 simulation that abstracts away memory hierarchy timing details to study adaptive cache line sizes for SPEC CPU92 and SPEC CPU95. They confirm findings of Inoue et al. [8] for SPEC CPU95, showing no single best line size exists. They then propose hardware that dynamically decrements virtual cache line size if fetched data are unused, and increment sizes if adjacent lines are cached. Replacement decisions are per line. Two bits per word track usage and signal presence/absence of adjacent lines, and they add memory per base cache line to track current virtual size. Our goals are similar, and our technique has lower overhead.

3 Design

In contrast to most previous superloading schemes, we optimize block size in the L2 cache. Generally, lower-level caches are relatively large, and thus exploit spatial locality more than upper-level caches, providing more potential benefit from variable-size logical blocks. Furthermore, changing the cache block size dynamically at L2 is likely less invasive and less complex due to higher cache access latency and often lower controller clock rate. Maintaining inclusion within the hierarchy is simplified if upper-level cache blocks are equal size or smaller than those at lower levels, as an upper level miss requires fetching at most one block from below. If L1 block sizes are adjustable, L2 blocks must be at least the maximum L1 logical block size, or else an L1 block size could exceed that of an L2 block for some configurations, thereby producing multiple misses in the L2 for a single L1 miss. Given the higher complexity and lower potential performance impact, we do not apply our mechanisms to L1 caches in this study.

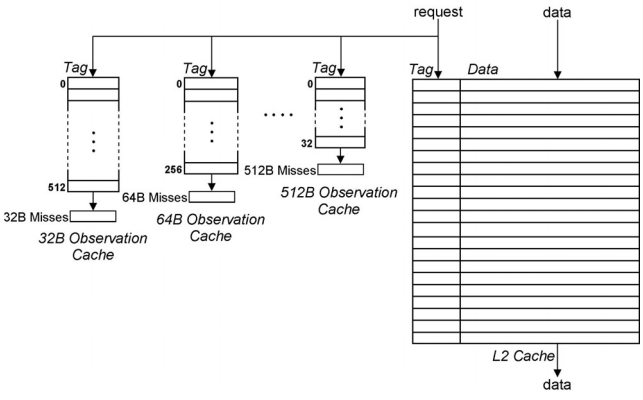


Fig. 1. Observation Cache Hardware

Adjusting on-chip cache block sizes introduces a feedback loop: a sensing unit observes spatial locality of executing code, and a controller tracks changes in locality characteristics, adjusting logical block size accordingly. On a miss, the cache controller fetches one or more physical blocks to load the logical block. Next we discuss the design rationale of these components, as well as their interaction.

3.1 Measuring Spatial Locality

To understand the locality sensor operation it is important to note that the conventional notion of spatial locality has both spatial and temporal aspects. First, caches can only exploit accesses to physically proximal data if that proximity matches the addressing scheme, i.e., if the data fall within a block. Second, references must occur before block eviction. The locality sensor presented here exploits these phenomena by monitoring performance of several small observation caches (OCs), each modeling a different block size. Each OC effectively tracks the number of accesses to an L2 block of a given size before that block is evicted. By comparing this metric for all allowed logical block sizes, the sensing unit determines optimal logical block size for the sequence of observed accesses. Given the inverse proportionality of hits and misses, OCs need only record misses.

Each OC maintains a miss counter along with state (tags, not data) of a few "recent" cache lines. OCs are accessed concurrently with the L2 cache, and are off the critical path. On each L2 access, the tag portion of the request is compared to the state held by each OC. On a miss, we evict an OC block and reuse it for the current request, incrementing the miss counter. The system maintains one OC per possible block size, and since each must cover the same capacity to make locality values comparable, OCs with smaller blocks have proportionally more entries. Figure 1 shows the organization of the observation caches. Since all OCs receive the same number of requests and represent identical capacities, the cache with fewest misses represents the L2 block size best exploiting program locality.

Locality values do not account for costs of larger logical cache blocks: they only express benefits in terms of cache hits over a given period.

3.2 Selecting Logical Block Size

Choosing the current optimal block size is left to system software: the clock interrupt handler periodically compares the miss counters of all OCs, resets the counters, and if appropriate, informs the cache controller of a new logical block size. This design increases flexibility and avoids fixing critical policy decisions in hardware. It eases performance debugging by making an application's spatial locality behavior visible to system software. Most importantly, it allows the OS to treat chosen block sizes as process state saved and restored on context switches, minimizing interference among processes in multiprogrammed environments.

A cache block twice the size can potentially exploit spatial locality twice as well, but it increases memory and bus occupancy/contention. For a fixed-size cache, larger blocks reduce the total number of blocks resident, which may increase conflict misses. Since the precise relationship between these factors is difficult to describe analytically, this work uses experimentally determined weights to account for the approximate cost of each potential block size. For the SPEC CPU2000 benchmarks, optimal weights are 1.0, 1.2, 1.8, 2.9, and 5.1 for blocks of 32B, 64B, etc., up through 512B. Note that weights used are identical for all benchmarks. Performance may improve for individual codes by further tuning weights, but this defeats our goal of providing a transparent mechanism. Analytic models for weight selection are part of ongoing work, and choosing the block size in software allows the weights to be adjustable.

After multiplying miss counts by the respective weights, the interrupt handler determines the minimum result and selects the corresponding block size. The chosen size is written to a control register that dictates how many physical blocks to fetch on a miss. In the system we study, the clock interrupt handler is invoked every 10 (simulated) milliseconds to determine the best block size for the application's next execution interval. Other context switch intervals would be interesting to investigate, but are beyond the scope of this work. On a context switch, the OS saves the current logical cache block size in the process control block just like other register values, and restores the next process's block size. Performance gains from adjusting block sizes dynamically must offset this small increase in context switch cost. Making logical block size part of the process context reduces interference between applications in multiprogrammed environments and avoids tuning intervals after context switches. Other implementations (see Sect. 5.3) could rely entirely on hardware, but we find the benefits of a hybrid hardware/software scheme to outweigh the small costs.

3.3 Dynamic Cache Line Size Adjustment

Our system adjusts L2 block sizes by grouping physical blocks into larger, logical cache blocks. On a miss, the controller fetches multiple physical blocks to effectively increase logical block size. When fetching the physical blocks, the cache

controller issues a sequence of requests to the memory controller via the system bus. The alternative, allowing different-sized bus requests, increases complexity of the bus protocol and memory controller while complicating coherence (since caches might need to snoop multiple physical blocks for any request, and may not be able to supply all data on a dirty hit if not all physical subblocks of a logical block are present).

Our experiments show parts of logical blocks are often already resident, and need not be fetched. Always fetching entire logical blocks may negatively affect bus utilization and memory performance. On the other hand, fetching logical blocks as a sequence of smaller physical blocks increases memory controller requests. Greater system bus utilization is offset by not having to fetch parts of logical blocks already in cache. Transmitting logical blocks as multiple requests has no impact on effective DRAM bandwidth, as long as DRAM page size and bank interleaving factors are no smaller than the largest logical block size. In this case, the memory controller can fully exploit the high bandwidth available from an open row.

Our design thus enjoys several advantages:

1. it minimizes impact of adjustable block sizes on cache design by minimizing required tag logic changes;
2. it allows multiple cache block sizes to coexist in cache (e.g., when applications share the cache, or during program phase transitions);
3. it avoids complications from partially resident logical blocks, and leaves bus and cache coherence protocols unchanged;
4. it allows the memory controller to exploit higher bandwidth of open DRAM rows (since requests for logical blocks are issued in rapid succession), as long as the bank interleaving factor is at least the maximum logical block size;
5. it benefits both instruction and data memory utilization, since it targets shared L2 caches;
6. it allows changing from any block size to any other, as opposed to enforcing fixed-size changes; and
7. it requires no profiling or user-level software or compiler intervention.

4 Evaluation

To evaluate the proposed design, we implement the observation caches (OCs), variable cache block size, and OS modules in the ML-RSIM system simulator [20], which models a dynamically scheduled processor with two cache levels, a SDRAM memory controller, and a number of I/O devices, such as a real-time clock and SCSI disk. The simulator executes SPARC binaries and runs a Unix-like NetBSD-based OS supporting Solaris-compatible system calls, process management, multiprogramming, and virtual memory [19]. Snooping and coherence are handled normally, since only logical, not physical, fetch sizes change.

We modify the simulator to implement the locality OCs and enhance the L2 cache controller to prefetch a programmable number of physical cache blocks on

Table 1. Baseline System Configuration

Parameter	Value
Processor	3.2 GHz dynamically scheduled, 6-wide fetch/decode/graduate, 80-entry ROB
L1 Instruction Cache	32 kB, 2-way set-associative, 32-Byte blocks, 2-cycle latency
L1 Data Cache	32 kB, 2-way set-associative, write-back, 32-Byte blocks, 2-cycle latency, dual-ported
L2 Cache	8 MB, 2-way set-associative, write-back, 32 to 512-Byte blocks, 24 cycle latency
System Bus	533 MHz, 8-Byte multiplexed address/data
Main Memory	266 MHz DDR-SDRAM, 4 physical banks interleaved at 512-Byte blocks
Operating System	NetBSD-based

a miss. This effectively changes the hardware prefetch distance, but does not change the cache. We modify the clock interrupt handler to read locality values, select the optimal block size, and write it to a control register that the context switch handler saves and restores. In our experiments, the OC for a 512-Byte block contains 32 entries, with each smaller cache containing proportionally more entries to capture the same logical working set. Hardware overhead for the OCs is only 3.31 kB of state. We investigate smaller OCs, finding sizes presented here to work best for this memory hierarchy. Compared to a 16-entry OC, a 32-entry version increases IPC, on average, by a negligible 2%, but by up to 22% for some floating point codes. A 16-entry OC may thus be a good design choice for platforms targeting integer codes. The OCs employ LRU replacement. We investigated other policies and find that LRU performs best. Since OCs need not support single-cycle access times, implementing LRU is feasible for these moderate-size, associative structures.

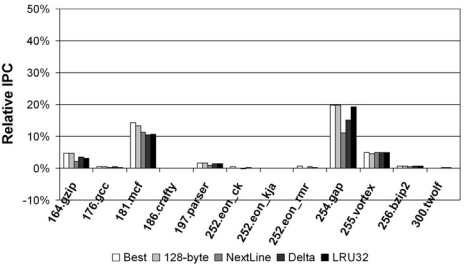
Our baseline represents modern or near-future workstation and server class systems, modeling an aggressive dynamically scheduled microprocessor with a conventional two-level cache hierarchy and DDR-SDRAM memory subsystem. Table 1 lists system parameters. We compare against both a simple next-line prefetcher and Nesbit and Smith's delta prefetcher [16]. Our scheme effectively behaves as an adaptable-distance, next-line prefetcher.

We present results for SPEC CPU2000 benchmarks run to completion with both training and reference inputs. Training datasets do not represent reference datasets, but shorter execution times make it feasible to observe *all* phases of program execution for a larger number of applications. Even short benchmarks show significant sensitivity to cache block size, confirming that they represent valid tools for initial evaluation of our scheme. Running to completion, despite long simulation times, allows us to evaluate the benefits of dynamic adjustment compared to fixed static sizes. We warm up the simulated file cache by reading all input files once; this avoids long-latency I/O operations during benchmark runs. Table 2 lists applications, datasets/configurations, and baseline execution times (in simulated seconds) within our model.

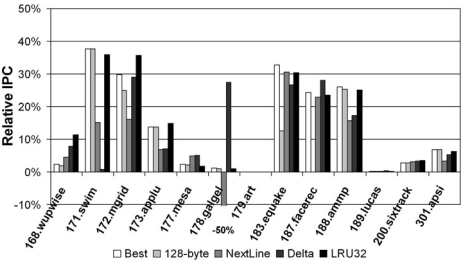
We exclude three applications from our training set runs: vpr and perlbmk require dynamic linking, which our simulation system does not support, and fma3d's excessive execution time makes it impractical to simulate to completion, even with the training dataset. We include all three input sets of 252.eon, since they yield different execution times. We evaluate 11 SPEC 2000 applications

Table 2. Benchmarks and Inputs

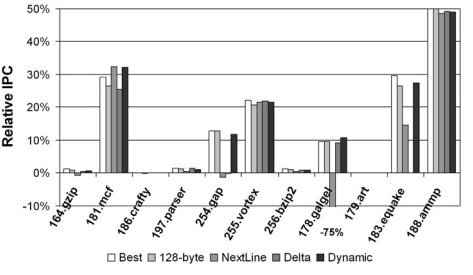
	Benchmarks	Training Inputs	Execution Times (sec)
Integer	164.gzip	input.combined	13.973
	176.gcc	cp-decl.s	1.765
	181.mcf	inp.in	6.160
	186.crafty	crafty.in	11.165
	197.parser	train	2.783
	252.eon	kajiya	3.907
	252.eon	cook	0.757
	252.eon	rushmeier	1.083
	254.gap	train.in -q -m 128M	2.942
	255.vortex	bendian.raw	4.523
	256.bzip2	input.compressed 8	12.963
	300.twolf	train	3.902
Floating Point	168.wupwise	wupwise.in	17.154
	171.swim	swim.in	5.948
	172.mgrid	mgrid.in	4.290
	173.applu	applu.in	4.611
	177.mesa	-frames 500 -meshfile mesa.in	14.253
	178.galgel	galgel.in	5.689
	179.art	-scanfile c756hel.in -trainfile1 a10.img -stride 2 -startx 134 -starty 220 -endx 184 -endy 240 -objects 3	1.080
	183.equake	inp.in	8.465
	187.facerec	train.in	10.223
	188.ammp	ammp.in	16.434
	189.lucas	lucas2.in	18.935
	200.sixtrack	inp.in	2.260
	301.apsi	apsi.in	2.817
		Reference Inputs	
Integer	164.gzip	input.source	23.197
	181.mcf	inp.in	71.642
	186.crafty	crafty.in	76.127
	197.parser	ref	111.37
	254.gap	ref.in -l ./ -q -m 192M	72.867
	255.vortex	bendian1.raw	32.298
	256.bzip2	input.source 58	25.088
Floating Point	178.galgel	galgel.in	50.692
	179.art	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10	11.249
	183.equake	inp.in	65.898
	188.ammp	ammp.in	62.778



(a)



(b)



(c)

Fig. 2. IPC of Best Static Block Size, Dynamic Block Size, and Hardware Prefetching relative to Worst Static Block Size for (a) Integer with Training Inputs, (b) Floating Point with Training Inputs, and (c) Reference Inputs

with reference inputs; others are omitted due to extremely long simulation times. These 11 applications have the shortest execution times and include seven integer and four floating point workloads. As shown in Sect. 5, floating point workloads tend to benefit more from our scheme, and thus its success on the reference inputs is understated.

Reading locality values, applying weights, and selecting a new block size incur a slight increase in clock interrupt-handler overhead. The additional code

increases interrupt handler cost on average by 150 cycles or 46.8 nanoseconds. Although this is minor, it must be offset by the performance gains from adjusting logical block size at runtime. All simulations in this study include a fully functional OS that accounts for the higher overhead.

5 Results

The simulation results confirm that the findings of Inoue et al. [8] and Veidenbaum et al. [24] in their L1 cache studies also apply to L2 caches: no single, statically selected block size is best for all applications. Overall, the best statically selected block size is 128 bytes and the worst is 32 bytes. This is based on the average relative slowdowns and speedups compared to best and worst statically selected block sizes for each application. Figure 2 shows performance improvement of the best static block size for each benchmark relative to the worst static block size, as well as the relative performance of our dynamic scheme, the best fixed block size (128 bytes), and the two hardware prefetching mechanisms (discussed in Sect. 5.2).

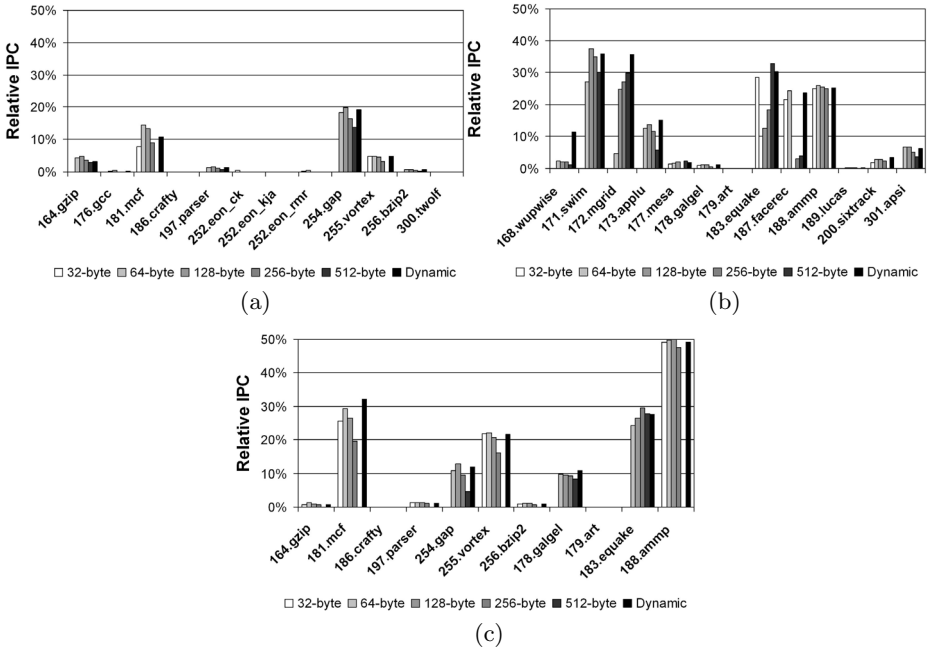


Fig. 3. IPC Improvement of Fixed Block Size and Dynamically Selected Block Size relative to worst case Static Size for (a) Integer with Training inputs, (b) Floating Point with Training Inputs, and (c) Reference Inputs

Figure 3 compares performance for the five statically selected block sizes vs. our dynamic scheme, all relative to the worst static block size for each benchmark. A single, fixed size is often inadequate, and common choices for particular

Table 3. Performance Details

Benchmark	Static Block Choices		Static Block Times (sec)		Next Line (sec)	Delta (sec)	Dynamic Time (sec)	Percent Difference			
	best	worst	best	worst				best vs. worst	nextL vs. worst	delta vs. worst	dyn vs. worst
164.gzip	128	32	13.973	14.661	14.357	14.144	14.199	4.69	2.07	3.53	3.15
176.gcc	128	512	1.765	1.773	1.768	1.767	1.768	0.45	0.28	0.34	0.28
181.mcf	64	512	6.16	7.186	6.373	6.443	6.422	14.28	11.31	10.34	10.63
186.crafty	64	256	11.165	11.17	11.164	11.167	11.166	0.04	0.05	0.03	0.04
197.parser	128	32	2.783	2.827	2.804	2.787	2.789	1.56	0.81	1.41	1.34
252.eon	32	64	0.757	0.760	0.762	0.757	0.759	0.39	-0.26	0.39	0.13
252.eon	32	64	3.907	3.909	3.912	3.907	3.909	0.05	-0.08	0.05	0.00
252.eon	64	128	1.083	1.089	1.085	1.087	1.089	0.55	0.37	0.18	0.00
254.gap	128	32	2.942	3.671	3.263	3.115	2.965	19.86	11.11	15.15	19.23
255.vortex	64	512	4.523	4.755	4.522	4.518	4.524	4.88	4.90	4.98	4.86
256.bzip2	128	32	12.963	13.047	12.999	12.959	12.965	0.64	0.37	0.67	0.63
300.twolf	64	32	3.902	3.903	3.897	3.897	3.902	0.03	0.15	0.15	0.03
168.wupwise	64	32	17.154	17.574	16.787	16.208	15.567	2.39	4.48	7.77	11.42
171.swim	128	32	5.948	9.533	8.103	9.456	6.114	37.61	15.00	0.81	35.86
172.mgrid	512	32	4.29	6.117	5.128	4.346	3.93	29.87	16.17	28.95	35.75
173.applu	128	32	4.611	5.343	4.972	4.971	4.542	13.70	6.94	6.96	14.99
177.mesa	512	256	14.253	14.588	13.865	13.841	14.325	2.30	4.96	5.12	1.80
178.galgel	64	512	5.689	5.753	8.619	4.169	5.693	1.11	-49.82	27.53	1.04
179.art	64	32	1.08	1.081	1.081	1.081	1.081	0.09	0.00	0.00	0.00
183.equake	512	64	8.465	12.603	8.753	9.253	8.778	32.83	30.55	26.58	30.35
187.facerec	64	128	10.223	13.493	10.406	9.711	10.307	24.23	22.88	28.03	23.61
188.ammp	64	512	16.434	22.212	18.716	18.388	16.616	26.01	15.74	17.22	25.19
189.lucas	64	32	18.935	18.988	18.935	18.928	18.935	0.28	0.28	0.32	0.28
200.sixtrack	64	512	2.26	2.324	2.251	2.248	2.241	2.75	3.14	3.27	3.57
301.apsi	64	32	2.817	3.023	2.924	2.863	2.836	6.81	3.27	5.29	6.19
Reference Inputs											
164.gzip	64	512	23.197	23.509	23.653	23.431	23.364	1.33	-0.61	0.33	0.62
181.mcf	64	512	71.642	101.303	68.522	75.257	68.661	29.28	32.36	25.71	32.22
186.crafty	64	128	76.127	76.146	76.131	76.135	76.132	0.02	0.02	0.01	0.02
197.parser	32	512	111.37	112.986	112.315	111.319	111.736	1.43	0.59	1.48	1.11
254.gap	128	32	72.867	83.544	84.617	83.916	73.682	12.78	-1.28	-0.45	11.80
255.vortex	64	512	32.298	41.402	32.475	32.368	32.487	21.99	21.56	21.82	21.53
256.bzip2	64	512	25.088	25.395	25.287	25.202	25.144	1.21	0.43	0.76	0.99
178.galgel	64	32	50.692	56.084	98.373	50.914	50.053	9.61	-75.40	9.22	10.75
179.art	32	256	11.249	11.25	11.249	11.249	11.249	0.01	0.01	0.01	0.01
183.equake	256	32	65.898	93.53	79.958	93.211	67.822	29.54	14.51	0.34	27.49
188.ammp	128	512	62.778	125.592	64.577	64.091	63.944	50.01	48.58	48.97	49.09

kinds of systems, e.g., embedded platforms or servers, may induce performance losses from "intuitively" selected sizes. Integer codes are less sensitive to block size changes: six of 10 show $\leq 1\%$ difference with training inputs, and three of seven show $\leq 1\%$ difference with reference inputs. Nonetheless, gzip, mcf, gap, and vortex show notable performance improvements (3.15%, 10.65%, 19.23%, and 4.86%, respectively, with training inputs, and 47.54%, 13.38%, and 27.44% for mcf, gap, and vortex with reference inputs). Differences between best and worst statically chosen sizes are much greater for many floating point applications: eight show significant speedups with training inputs (wupwise, swim, mgrid, applu, equake, facerec, ammp, and apsi, by 11.42%, 35.88%, 35.79%, 15.05%, 30.35%, 21.32%, 25.19%, and 6.25%, respectively) and three of the four show significant speedups with reference inputs (10.75%, 27.49%, and 49.09% for galgel, equake, and ammp, respectively). Table 3 shows detailed results for all applications.

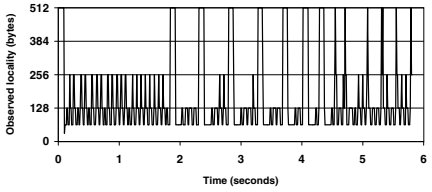


Fig. 4. Spatial Locality for galgel

Table 4. Delta HW Prefetcher Parameters

Parameter	Value
Index Table entries	512
Index Table replacement	FIFO
History Table entries	512
Prefetch Width	4
Prefetch Depth	4

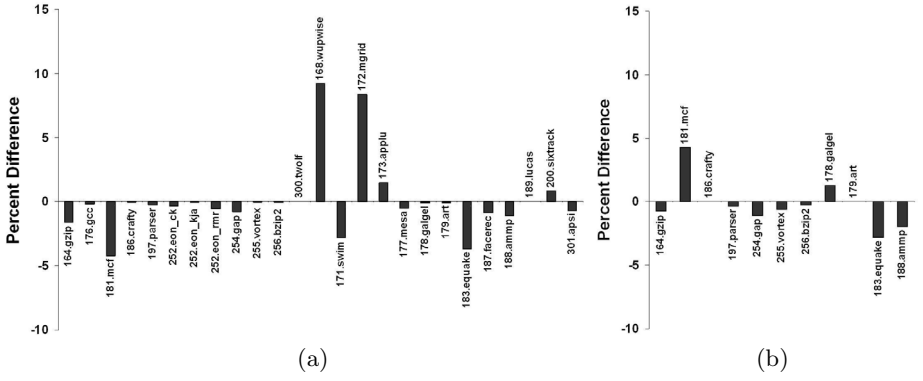


Fig. 5. Best Static vs. Dynamic Blocksize for (a) Training and (b) Reference Inputs

5.1 Phase Adjustment

To illustrate the presence and magnitude of locality phases, Fig. 4 shows the time line of preferred cache block sizes and corresponding spatial locality for galgel. Such locality changes cause some applications to perform better using dynamic selection over their optimal static block size. Other applications perform better with a fixed block size due to very short phases (as in gcc), in which case the dynamic mechanism is always "one step behind" due to its adjustment latency, or lack of phase behavior (as in twolf).

Figure 5 compares the performance of the best statically chosen block size per application vs. our dynamic scheme. In six training cases and two reference sets our scheme outperforms the best static block size, ranging from negligible amounts to almost 10%. In other cases, the dynamic scheme performance lags slightly behind the best static block size, usually by a negligible amount, and always by less than 5%. Advantages of dynamic selection are that it requires no software intervention, needs no profile-directed feedback, and enjoys low hardware complexity. Most importantly, since no fixed block size is optimal for all applications, a dynamic scheme can detect appropriate block sizes at runtime.

For floating point applications, workloads with small differences between best and worst static block sizes tend to have small performance differences. The exception is apsi, for which the difference in best and worst block size is a single increment, but for which the percentage difference in performance is 6.81%.

Obviously, this application will benefit from adaptable block sizes, even though the desired size changes little. Applications whose best and worst block sizes differ by at least two powers of two tend to exhibit larger performance differences.

The rightmost columns in Table 3 show percentage performance improvement of the different schemes relative to the *worst* statically selected block size. We highlight cells where these values round to 5% or more. The rightmost column shows the improvement for dynamically selected block sizes, and the column fourth from the left shows improvement for the best-performing statically selected blocks over worst-performing statically selected blocks. These data argue for adaptable architectures: although not all applications require different block sizes, performance ramifications are significant for those that do. Interestingly, these data contradict that of Chen et al. [2] in their study of L1 block sizes: we find that 179.art, 254.gap, and 172.mgrid benefit from larger blocks, as do 171.swim, 173.applu, 183.equake, and 301.apsi. Differences in results are likely due to our targeting different levels of memory and to differences in simulation methodology (we model whole applications with the OS, whereas they perform arbitrary partial execution of 500 million instructions).

5.2 Prefetching

In an attempt to hide increasing memory latencies, hardware prefetching mechanisms have received extensive study. Hardware prefetchers read blocks a program is likely to access before demand misses occur. Block predictions tend to be based on the program's recent memory access history. We compare our dynamic block size scheme against two hardware prefetching schemes: a simple next line prefetcher and a delta predictor that uses the global history buffer of Nesbit and Smith [16]. The latter predictor yields some of the best performance gains to date for SPEC2000. Table 4 gives parameters for this predictor (see Nesbit and Smith [16] for parameter descriptions).

On the whole, both the next line and delta prefetchers perform worse than both our dynamic block scheme and the best static block size. Figure 2 shows performance improvement of the prefetching mechanisms relative the worst static block size. OC-guided dynamic block sizes achieve 5.6% and 2.3% higher IPC, on average, than next-line and delta prefetching, respectively. For the floating point workloads, dynamically sized blocks achieve up to 49.1%—and on average 10.3%—higher IPC than next-line prefetching, and up to 35.3%—and on average 3.3%—higher IPC than delta prefetching. For integer benchmarks, prefetching performs as well as our dynamic and the best static block methods: integer applications are less sensitive to block size, as evidenced by the small differences in performance of the best and worst static block sizes in Fig. 2. The exception is gap, for which our dynamic approach attains 9.1% and 4.8% higher IPC with training inputs and 12.9% and 12.2% higher IPC with reference inputs than the next-line and delta prefetchers, respectively.

Even more important than raw IPC improvement is the performance stability provided by our dynamic scheme. Hardware prefetching schemes tend to be inconsistent in their ability to improve an arbitrary application. Sometimes they

Workload	Programs	Total Runtime (s)
(a) Training Inputs		
Batch 1	172.mgrid, 255.vortex, 173.applu	14.25
Batch 2	181.mcf, 183.equake	24.46
Batch 3	164.gzip, 177.mesa	28.08
Batch 4	172.mgrid, 254.gap	7.65
(b) Reference Inputs		
Batch 5	183.equake, 188.ammp	237.32

Fig. 6. Multiprogramming Workloads

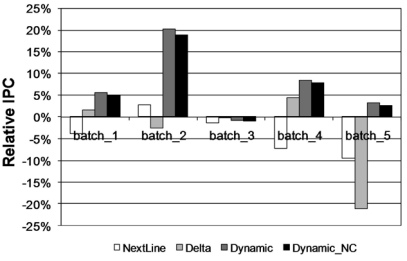


Fig. 7. Multiprogramming Performance Relative to Best Static Block

provide drastic improvements, even outperforming our dynamic approach or the best static cases, but other times they perform very poorly—sometimes even worse than the worst static case. Our dynamic scheme, on the other hand, never lags behind the best static case by more than 5%. This consistency is highly desirable for systems designed to run a wide variety of applications.

5.3 Multiprogramming

An advantage of our dynamic block size adjustment scheme is the ability to save and restore the preferred block size on a context switch. Treating the logical block size as part of the process context avoids the overhead of retraining the locality caches after a context switch. This subsection compares performances of five multiprogrammed workloads for static and dynamic block sizes.

As in standard Unix OSs, the kernel running on the simulated system maintains time quanta per process, performing a context switch every 100 milliseconds if there is another process in the ready queue. Each workload consists of two or three SPEC CPU2000 applications of similar execution time (see Table 6). Applications in each workload prefer different block sizes, stressing the dynamic adjustment scheme.

Figure 7 shows the performance improvement for these workloads for the two hardware prefetching mechanisms, and two variations of the dynamic scheme relative to the performance of the *best* static block size. In four of five workloads, dynamic block size adjustment outperforms the best static block size, since the dynamic scheme adjusts to spatial locality variations both within and across applications. The dynamic scheme outperforms the two prefetching methods for the same four workloads. The right-most bars in each set, labeled "Dynamic/NC", correspond to experiments in which the OS does not save/restore logical block sizes on context switches. In this case, the dynamic scheme requires an additional 10 ms to select the appropriate logical block size. Given a 100 ms time slice, the system operates for approximately 10% of time with a potentially suboptimal logical block size. Advantages are lower context-switch overhead and avoiding modification of often highly optimized context switches. On the other hand, saving/restoring optimal block sizes only requires adding

one field to process control blocks and adding one load/store to the save/restore context switch code.

Saving and restoring logical block size performs better by 0.5-1.1%. If we had not modeled a full system, with kernel, we might dismiss such results. Nonetheless, given our methodology, we believe our findings warrant further investigation. The ability to resume execution with previously selected logical block sizes compensates for slight increases in context switch cost. Quantifying software engineering costs of arguably small changes to two pieces of the kernel is difficult, and performance gains may or may not be justified depending on circumstances.

Most important, dynamic block sizes can significantly improve performance over a single static block size. The dynamic scheme outperforms even the best statically selected block size by 3-20% for four of the five multiprogrammed workloads, confirming that the approach represents a viable alternative to software-directed techniques.

6 Conclusions

This paper presents a hardware-centric technique to adjust logical L2 cache block sizes at application runtime. A set of observation caches (OCs) measure spatial locality relative to each other and make the values available to the clock interrupt handler, which selects the best logical block size for the next period. In addition to providing greater flexibility in terms of adjustment policy, this combined hardware/software approach exposes spatial locality behavior to software for performance debugging and profiling, and lets the OS treat the logical block size as part of the process state on a context switch. A hardware adaptation of the policy decision is straightforward for cases where OS modifications are undesirable or infeasible.

Detailed system simulation results using SPEC CPU2000 confirm that our technique approaches performance of the optimal statically selected block size for most benchmarks, and outperforms it in many cases. The approach is up to 32% and 49% better than the worst block size choice for integer and floating point applications, respectively. We never degrade performance over the *worst* statically selected block size, and never by more than 5% over the *best* statically selected block size. *Note that these comparisons are to the worst and best cases, not to what one would see in real systems with fixed block sizes. Fixed block sizes rarely match the best block size for the applications we study.* Speedups are less dramatic than other spatial locality prediction schemes for some applications, but our approach's simplicity and performance stability recommend it. We deliver robust performance without compiler or application intervention, and the approach is thus transparent to user-level software. Our scheme performs especially well on multiprogrammed workloads, since it can select the best block size for each individual benchmark. This allows it to significantly outperform the best statically selected block size for an application "class" that is typical in many computing environments.

References

1. Calder, B., Krintz, C., John, S., Austin, T.: Cache-conscious data placement. In: Proc. 8th ACM Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 139–149 (October 1998)
2. Chen, C., Yang, S., Falsafi, B., Moshovos, A.: Accurate and complexity-effective spatial pattern prediction. In: Proc. 10th IEEE Symposium on High Performance Computer Architecture, pp. 187–276 (February 2004)
3. Chilimbi, T., Davidson, B., Larus, J.: Cache-conscious structure definition. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 13–24 (May 1999)
4. Dahlgren, F., Dubois, M., Stenstrom, P.: Fixed and adaptive sequential prefetching in shared memory multiprocessors. In: Proc. International Conference on Parallel Processing, pp. 733–746 (August 1993)
5. Ding, C., Kennedy, K.: Improving cache performance in dynamic applications through data and computation reorganization at run time. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 229–241 (May 1999)
6. Gonzalez, A., Aliagas, C., Valero, M.: A data cache with multiple caching strategies tuned to different types of locality. In: Proc. 1995 International Conference on Supercomputing, pp. 338–347 (1995)
7. Gornish, E., Veidenbaum, A.: An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. In: Proc. International Conference on Parallel Programming, pp. 35–70 (August 1994)
8. Inoue, K., Kai, K., Marakami, K.: High bandwidth variable line size cache architecture for merged DRAM/logic LSIs. IEICE Transactions on Electronics 81(9), 1438–1447 (1999)
9. Johnson, T., Merten, M., Hwu, W.: Run-time spatial locality detection and optimization. In: Proc. IEEE/ACM 30th International Symposium on Microarchitecture, pp. 57–64 (December 1997)
10. Kane, G.: MIPS RISC Architecture. Prentice Hall, Englewood Cliffs (1989)
11. Kodukula, I., Pingali, K.: Data-centric transformations for locality enhancement. International Journal of Parallel Programming 29(3), 319–364 (2001)
12. Kumar, S., Wilkerson, C.: Exploiting spatial locality in data caches using spatial footprints. In: Proc. 25th IEEE/ACM International Symposium on Computer Architecture, pp. 357–368 (June 1998)
13. Leung, S.: Array Restructuring for Cache Locality. PhD thesis, University of Washington (August 1996)
14. McKinley, K.S., Temam, O.: Quantifying loop nest locality using SPEC 1995 and the perfect benchmarks. ACM Transactions on Computer Systems 17(4), 288–336 (1999)
15. Mellor-Crummey, J., Whalley, D., Kennedy, K.: Improving memory hierarchy performance for irregular applications using data and computation reorderings. International Journal of Parallel Programming 28(3) (June 2001)
16. Nesbit, K., Smith, J.: Data cache prefetching using a global history buffer. In: Proc. 10th IEEE Symposium on High Performance Computer Architecture, pp. 96–105 (February 2004)
17. Pingali, V., McKee, S., Hsieh, W., Carter, J.: Restructuring computations for temporal data cache locality. International Journal of Parallel Programming 31(4), 306–338 (2003)

18. Pugh, W., Rosser, E.: Iteration space slicing for locality. In: Carter, L., Ferrante, J. (eds.) LCPC 1999. LNCS, vol. 1863, pp. 164–184. Springer, Heidelberg (2000)
19. Schaelicke, L., Davis, A., McKee, S.: Profiling interrupts in modern architectures. In: Proc. 8th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 115–123 (July 2000)
20. Schaelicke, L., Parker, M.: ML-RSIM home page (May 2005), `http://www.cs.utah.edu/~lambert/mlrsim/`
21. Seznec, A.: Decoupled sector caches. IEEE Transactions on Computers 46, 210–215 (1997)
22. Temam, O., Granston, E., Jalby, W.: To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In: Proc. Supercomputing 1993, pp. 410–419 (December 1993)
23. Temam, O., Jegou, Y.: Using virtual lines to enhance locality exploitation. In: Proc. 8th ACM International Conference on Supercomputing, pp. 344–352 (July 1994)
24. Veidenbaum, A., Tang, W., Gupta, R., Nicolau, A., Ji, X.: Adapting cache line size to application behavior. In: Proc. 13th ACM International Conference on Supercomputing, pp. 145–154 (1999)
25. Vleet, P.V., Anderson, E., Brown, L., Baer, J.-L., Karlin, A.: Pursuing the performance potential of dynamic cache line sizes. In: Proc. International Conference on Computer Design, pp. 528–537 (October 1999)
26. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. ACM SIGPLAN Notices 26(6), 30–44 (1991)
27. Wolfe, M.: Optimizing Supercompilers for Supercomputers. MIT Press, Cambridge (1989)

ACM: An Efficient Approach for Managing Shared Caches in Chip Multiprocessors

Mohammad Hammoud, Sangyeun Cho, and Rami Melhem

Department of Computer Science
University of Pittsburgh
{mhh,cho,melhem}@cs.pitt.edu

Abstract. This paper proposes and studies a hardware-based adaptive controlled migration strategy for managing distributed L2 caches in chip multiprocessors. Building on an area-efficient shared cache design, the proposed scheme dynamically migrates cache blocks to cache banks that best minimize the average L2 access latency. Cache blocks are continuously monitored and the locations of the optimal corresponding cache banks are predicted to effectively alleviate the impact of non-uniform cache access latency. By adopting migration alone without replication, the exclusiveness of cache blocks is maintained, thus further optimizing the cache miss rate. Simulation results using a full system simulator demonstrate that the proposed controlled migration scheme outperforms the shared caching strategy and compares favorably with previously proposed replication schemes.

1 Introduction

Advances in process technology have enabled integrating billions of transistors within a single chip. These advances combined with an incessant need to improve computer performance paved the road to the emergence of chip multiprocessors (CMPs). Chips capable of small-to-medium scale multiprocessing are commercially available [15, 21] and platforms having more cores are forthcoming [13].

As the realm of CMP is continuously expanding, it must provide high and scalable performance. The constantly widening processor-memory speed gap will substantially increase the capacity pressure on the on-chip memory hierarchy. The lowest-level on-chip cache not only needs to utilize its limited capacity effectively, but also has to mitigate the increased latencies due to wire delays [8]. Accordingly, a key challenge to obtaining high performance from CMP architectures is to manage the last level cache so that the access latency is reduced effectively and the capacity utilized efficiently.

CMP caches are typically partitioned into multiple banks for reasons of growing wire resistivity, power consumption, thermal cooling, and reliability considerations. Non-Uniform Cache Architecture (NUCA) has been employed to organize the resultant multiple cache banks [4, 10, 14, 24, 25]. One common practice for NUCA is the shared scheme where cache banks are all aggregated to form a logically shared cache [4, 15, 21, 24, 25]. Each memory block is mapped to a unique cache set in a unique cache bank offering thereby high cache capacity utilization. Unfortunately, shared caches have a latency problem. A cache block may reside at a bank far away

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 355–372, 2009.

from the requester core. It is a challenge to maintain the advantage (cache capacity) of the shared cache CMP design and preclude the disadvantage (access latency).

Block replication and migration have been suggested as techniques for shared CMP caches to tackle the latency problem by frequently copying or moving accessed blocks to cache banks closer to the requesting processors [3, 6, 7, 8, 15, 16, 17, 21, 25]. Replication in general results in reduced cache hit latencies but may degrade cache hit rate. In fact, blind replication can be detrimental since the capacity occupied by replicas could increase significantly resulting in performance degradation [3]. Migration, on the other hand, maintains the exclusiveness of cache blocks on chip and preserves the high utilization of the caching capacity. Furthermore, it maintains the simplicity of the underlying cache coherence protocol. However, migration has been shown to be less effective for CMP caches than for uniprocessors [4, 16]. The issue in the CMP domain is that migration in multiple directions can cause migration conflicts, with shared blocks ping-ponging between processors [14]. Besides, locating migratory blocks in bank sets may turn out to be very expensive to an extent that it offsets the benefits offered by the migration technique.

We demonstrate through an example the difficulty behind block migration and the inefficiency it may cause with shared L2 cache design in the CMP context. Figure 1(a) illustrates a 16-core tiled CMP. We assume a shared scheme where L2 cache slices are logically shared among all tiles. Upon an L2 miss, a line is fetched from the main memory and placed in a home tile determined by a subset of bits of the line's physical address. The figure shows a case where a block, B, has been originally requested by tile 3 and mapped to tile 6. Later tiles 0 and 8 request the same cache block B. Tile 3 incurs 6 network hops, computed as twice the Manhattan distance between the requester and the target tiles (dimension-ordered routing [20]), to reach the home tile and satisfy its request. Tiles 0 and 8 incur 8 hops each to satisfy their requests.

Figure 1(b) illustrates a naïve first touch migration policy that directly migrates B to the original requester. Employing that, tile 3 will save the 6 network hops when touching B for the second time, assuming that it checks its local L2 tags before accessing the home tile. Block B, however, has been pulled away from the other two sharers incurring additional 6 hops for each one to locate the block on its new host tile. Consequently, even though one tile made a gain, in total there is a loss of 6 network hops. Besides, the on-chip network traffic increases due to the three-way cache-to-cache communications

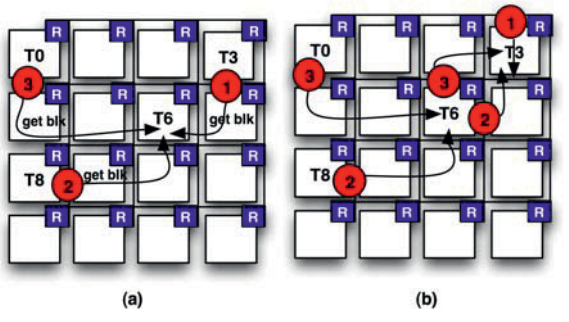


Fig. 1. (a) The Original Shared CMP Scheme. (b) A Simple Migration Example.

to satisfy sharers' requests. Specifically, when tile 0 (or 8) requests B, it has to check with tile 6 first, which is B's home tile, before its request is redirected to tile 3.

Though the above example shows a naïve migration policy, it highlights the intuition that applying migration in a non-controlled fashion to NUCA designs can lead to performance degradation. The problem, in fact, is that the best host of a cache block is not known ahead of time. Thus migration or replication interferes to rectify the situation. It would be highly beneficial if there is a mechanism that can dynamically and adaptively locate the best host on chip for each cache block, and move consecutively the block to that host without incurring undesirable implications.

This paper grants a fresh thought to the data migration technique as a way to manage shared CMP caches and studies its effectiveness in tiled CMPs. We propose a novel hardware-based *adaptive controlled migration* (ACM) mechanism that relies on prediction to collect information about which tiles have accessed a block and then, assuming that each of these tiles will access the block again, dynamically migrates the block to a tile that minimizes the overall number of network hops needed. Simulation results demonstrate the effectiveness, scalability, and stability of the proposed scheme using a variety of workloads that exhibit *no-sharing*, *little sharing*, or *sharing* of cache blocks.

The contributions of the ACM mechanism are as follows:

- It demonstrates that migration, if done in an adaptive controlled fashion, yields an average L2 access latency that is on average 20.4% better than the nominal non-uniform shared L2 cache scheme, and 20.8% better than a conventional replication strategy for the simulated benchmarks.
- The proposed mechanism demonstrates the effectiveness of migration in the CMP domain and opens new research opportunities and directions for computer architects.
- The proposed mechanism avoids replication of cache blocks and reduces the overall L2 cache access latency without degrading the cache miss rate.

The rest of the paper is organized as follows. Section 2 delves more onto the idea of ACM and explains it in detail. Section 3 presents the ACM microarchitecture and hardware cost. Section 4 discusses our evaluation methodology and presents a quantitative assessment of our proposed mechanism. Other related works are recapitulated in Section 5, and Section 6 concludes.

2 Adaptive Controlled Migration

2.1 Baseline Architecture

This study assumes a tiled CMP model similar to the one depicted in Figure 1 and appeared many times in literature [9, 24, 25]. Tiled CMPs scale well to larger processor counts and can easily support families of products with varying numbers of tiles, including the option of connecting multiple separately tested and speed-binned dies within a single package [24]. The CMP model is organized as a 2D array of replicated tiles each with a core, a private L1 cache, an L2 bank, a directory for L1 coherence

maintenance, and a switch that connects the tile to the on-chip network. The L2 banks form a logically shared L2 cache. For the L1 level coherence enforcement, a distributed directory-protocol is modeled. The directory is distributed among all the tiles by mapping each memory block to a *home tile* [16, 18, 25]. On an L2 miss, a block is fetched from the main memory and mapped to its home tile determined by a subset of bits of the physical address called the home select (HS) bits. Consecutively the fetched block is copied to the L1 cache of the requester core. If any other core requests the same cache block at a later time, the home tile is accessed and the cache block is copied to its L1 cache. Cores that maintain copies of a certain cache block in their L1 caches are all denoted as *sharers* of this block.

A cache block may map to a tile that is close to or far away from the requester core. Therefore, the model introduces a NUCA design where accesses to any L2 bank varies depending on the network congestion and number of network hops between the requester and the target tiles. We assume a mesh network topology with dimension-ordered (XY) routing [20] where packets are first routed in the X and then the Y direction. Therefore, the number of network hops can be computed as twice the Manhattan distance between the requester and the target tiles.

2.2 Predicting Optimal Host Location

Keeping a block in its home tile is often sub-optimal. Ideally, we want to place a cache block in the tile that best optimizes the overall latencies. However, the best host tile for a block is not known until runtime because many cores may compete for that block. Consequently, a dynamic adaptive mechanism that monitors the runtime accessibility of a block and makes a decision about the best location for the block is needed.

We propose a simple location algorithm that attempts to locate the optimal host of a cache block at runtime and designates it as its new *host tile*. It computes the *total latency cost* for a given cache block on each of the potential hosts and chooses the *minimum*. In order to achieve this, the algorithm keeps some runtime information, particularly, a pattern for the accessibility of the cache block. The pattern is essentially a bit vector to indicate whether the block has been accessed by a specific core or not. It can be built at run time with different *migration frequency levels*. The migration frequency level is the number of times a block is accessed before attempting to migrate it. Whenever a core accesses a block, its corresponding bit is set in the associated bit vector and a *use counter* associated with the block is incremented. This continuously shapes up an accessibility pattern for the given block and provides the aspired runtime information. When the use counter reaches the specified migration frequency level, the location algorithm interferes and selects a new host for the block that minimizes the total L2 access latency for all the sharers identified by the accessibility pattern. The pattern and the use counter are both cleared when the block is migrated so as to initiate a new pattern construction. This is a simple prediction scheme that depends on the past to predict the future. A core that accessed a block in the past is likely to access it again in the future. Because our algorithm makes its decision based on this pattern, we call the located host a *predicted optimal host*.

To exemplify how the ACM mechanism works, Figure 2 portrays 4 different cases for potential hosts that a cache block may migrate to. S stands for a sharer and H for

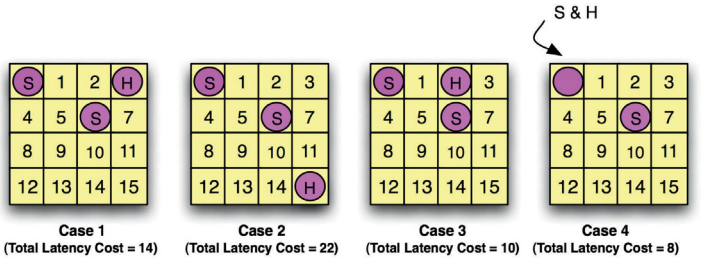


Fig. 2. An Example of How ACM Works (S = Sharer, H = Host)

the current host. A sharer is a tile that accessed the block in the past. A host is a tile that is currently hosting the cache block. The designated block has two sharers, tiles 0 and 6. The block can be potentially hosted by any of the 16 tiles, but in Figure 2 we depict only four cases where the tiles 3, 15, 2, and 0 host the block. Total latencies of 14, 22, 10, and 8 network hops are incurred by the sharers to locate the data block in these four designated hosts respectively. Among these, host 0 gives the minimum aggregate latency and is selected by the ACM mechanism to be the predicted optimal host.

2.3 Locating Migratory Blocks and Cache the Cache-Tags Policy

After locating the new predicted optimal host for a cache block, migration is performed. However, a question that follows is: how can a sharer later locate a cache block that is no more at its original home tile but migrated to a different tile? Zhang and Asanović [24] proposed an extra array of tags per tile to keep track of the locations of migratory blocks. Specifically, when a block, B, is migrated for the first time out of its home tile, an entry for B, serving as a short *fingerprint*, is allocated in this extra tag array at the home tile to point to the new location of B. Later if a sharer S reaches the home tile of B and fails to find a matching tag in the regular L2 tag array but hits in the extra tag array, the current host of B, pointed out by the matched fingerprint, satisfies the request. Specifically, data is forwarded to S from the current host of B using three-way cache to cache communication. If S fails to find a matching tag in both the regular and the extra tag arrays, an L2 miss is reported and data is fetched from the main memory. If S hits in the regular L2 tag array, then the L2 CMP shared protocol is simply followed. Figure 3(a) illustrates how a sharer can locate a cache block, B, that has already been migrated. This migration policy saves latency only for the tile to where B has been migrated. For other sharers, it fails to exploit *distance locality*. That is, the request may incur significant latency to reach the home tile even though the data is located in close proximity. Furthermore, it causes extra on-chip network traffic.

ACM uses a similar data structure found in [24] but with a simple, yet essential extension. We name this extended data structure the Migration Table (MT table). The idea is to *cache the cache-tags* in the MT table. Specifically, the MT table of a tile T can now hold two types of tags: (1) a tag for each block B whose home tile is T and had been migrated to another tile, and (2) tags to keep track of the locations of the migratory blocks that have been recently accessed by T but whose home tile is not T. We refer to the first type as *local MT tags* and to the second as *remote MT tags*.

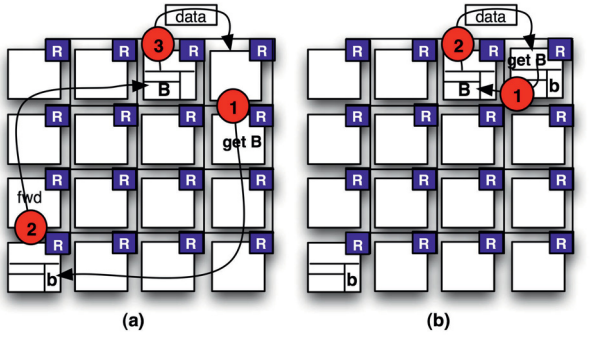


Fig. 3. Locating a Migratory Block B (a) Without *Cache the Cache-Tags* Policy. (b) With *Cache the Cache-Tags* Policy.

Whenever a sharer issues a request to a block, its local MT table is checked first for a matching tag. On a miss, the home tile is reached. If the block at the home tile has already been migrated, it is located in its new host and its tag is further cached in the requester's MT table, so as to reduce latency for subsequent accesses. If the block on the home tile was not migrated yet, the usual L2 shared protocol is followed. If the requester core hits in its local MT table, the corresponding block is directly retrieved from the location that the local tag designates; that is, the tile currently hosting the block. This is done without any need to visit the home tile. Accordingly, the three-way cache to cache communication problem exposed by the proposal in [24] is solved by the cache the cache-tags policy upon hits in the MT table. Figure 3 (b) shows how the cache the cache-tags idea solves the 3-way communication problem. The sharer finds a matching tag in its local MT table and quickly locates the data block on its host. Consequently, the profit gained by the migration technique is maintained, and the on-chip network traffic is improved.

The remote and local MT tags need to be kept consistent. That is, each time the block migrates, the corresponding remote and local MT tags in the MT tables need to be updated to indicate the block's new located host. We accomplish this by embedding a directory state in each of the MT tags. This directory state acts as a bit mask to indicate which tile has cached a remote MT tag for the associated cache block. It is simply a bit vector with a single bit corresponding to each core. Whenever a core caches a tag, its corresponding bit is set in the bit vector augmented with the local MT tag at the home tile. Effectively, each MT tag, or *MT entry* hereafter, is composed of four components: (1) The tag of the migratory cache block, (2) a bit vector that acts as a directory to maintain consistency of multiple copies of MT entries cached in different MT tables, (3) a bit to specify whether the entry is remote or local, and (4) an ID that points to the tile that is currently hosting the cache block. Hence, whenever a cache block migrates, its corresponding MT entry at the home tile is accessed and the ID is updated to point to the new host location. Furthermore, the cached remote MT entries, identified by the augmented bit vector, are all updated to indicate the new block's host. Note that the home tile that is to be reached to update the local MT entry is known via checking

the HS bits of the given physical address of the L2 request that triggered the migration of the L2 block. Accordingly, we need not store any backward pointer to locate the home tile.

As each associative set of blocks in the MT table contains a combination of local and remote MT tags, it is wise to never evict a local MT tag in favor of a remote one. This is because the local MT tag may have many active sharers. It may further have multiple associated remote MT tags stored at sharers' MT tables. Consequently, replacing it calls for eviction of the data block itself and all its associated remote tags. To avoid potential performance degradation, the MT table replacement policy replaces the following two classes of tags in descending order: (1) an invalid tag, (2) the least recently used remote MT tag. For now, we assume that the size and the associativity of the MT table are identical to those of the regular L2 tag array. In Section 4 we study the sensitivity of the ACM mechanism to different MT table sizes.

2.4 Replacement Policy Upon Migration: Swapping the LRU Block with the Migratory One

After the location algorithm designates a new host for a block B and the migration is to be performed, a decision must be made about which block to replace in the new host T located for B. If there is no invalid block in the target set at T, a naïve approach would replace the LRU block, say, D. However, because cache accesses might not be well distributed over the cache sets, there could be a capacity pressure at T, and D could be requested again. Hence, we try not to discard block D but to swap it with B so as to maintain the copy on chip. B and D could be migratory or non-migratory blocks. Migratory blocks are those that already migrated out of their home tiles while non-migratory ones are those that have not migrated yet. If B is non-migratory, a local MT entry should be allocated at its home MT table. If no entry is found to be replaced at the MT table, as planned by the MT replacement policy, migration is not performed. If B could be migrated and D is a migratory block, then they are simply swapped and D's associated MT entries are all updated to expose the change of the host location. If D is non-migratory, its local MT table is checked for a valid entry to replace. If a valid entry is found, a local MT entry is allocated and B and D are swapped. If no valid entry is found then D is simply discarded and B migrates to the new located host. Of course, if B also is a migratory block then when migrated, all its associated MT entries are updated to denote its new host.

Such a swapping policy is, in fact, very effective and robust that it makes our scheme applicable even to workloads that don't share cache blocks. We illustrate this effectiveness and robustness via an example. Figure 4 depicts a case for a single thread that exhibits *no sharing* at all and runs on tile 3 (the microarchitecture of the portrayed tiles is discussed in Section 3). We assume that the thread's working set is too large to fit entirely in the L2 cache bank of tile 3. To reduce L2 access latency, and with migration frequency level of 1, our location algorithm will choose to migrate all requested blocks to the L2 bank of tile 3 after accessing each for the second time. Figure 4(a) depicts the placement of block B after it has been requested by tile 3 and mapped to the home tile 5 (HS of B = 0101). Figure 4(b) demonstrates the migration of B to the L2 bank of tile 3 after tile 3 accessed B for the second time. Note that a local MT tag, b, is allocated in

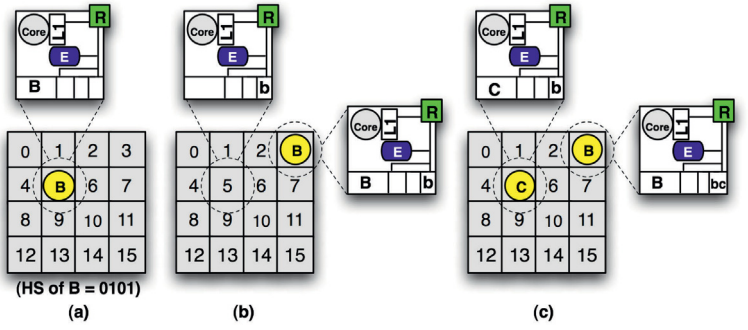


Fig. 4. An Automatic Data Attraction Case offered by ACM

the MT table at the home tile 5 and a remote one allocated at tile 3, as planned by the cache the cache-tags policy.

The case shows the capability of the ACM mechanism to automatically attract data to local tiles, thus allowing cores to access blocks very fast for subsequent requests. However, ACM is robust enough that it doesn't allow such *automatic data attraction* to continue freely and blindly, potentially causing increase in L2 miss rate. The *swapping with the LRU* policy suggests that when no invalid block at the target set of the located host T is found (capacity pressure), an attracted block B is swapped with the LRU block at T, thus avoiding increase in L2 miss rate. Figure 4(c) assumes that when B has been migrated to tile 3 it produces capacity pressure. The LRU block, C, at tile 3 is accordingly swapped with B as planned by the swapping with LRU policy, thus maintaining C on chip and accordingly avoiding any increase in the L2 miss rate. Note that a local MT tag c is allocated in the MT table at tile 3 as planned by the cache the cache-tags policy.

The case in fact demonstrates some resemblance to the victim replication strategy [25]. Victim replication also automatically attracts data to local tiles upon L1 evictions in order for subsequent accesses to save latency. However, it doesn't provide control on the capacity usage and can greatly reduce the available caching space. Section 4 presents a comparison between ACM and the victim replication scheme.

Finally, different cache blocks may experience entirely different degrees of sharing over time and demonstrate diverse access patterns. The ACM algorithm collects those non-uniform access patterns and based on them, finds a suitable location for data blocks that best minimize the L2 access latency. Thus ACM inherently doesn't prefer any specific on-chip tile over the other. However, if at any course of execution, a cache bank receives a capacity pressure more than other banks (similar to the above case), ACM robustly relaxes the pressure via the swapping with the LRU policy.

2.5 Discussion: Cache Blocks in Transit

The ACM mechanism can easily preclude any potential for *false misses* which occur when an L2 request falsely fails to hit a cache block because it is in transit from one bank to another. When migration is to be performed, a copy B' of the cache block B is kept at the current bank so as if an L2 request arrives while B is in transit, the request

is immediately satisfied without incurring any delay. When B reaches the new host, an acknowledgment message is sent back to the old host to discard B'. The old host keeps track of any tile that accesses B', and when receiving the acknowledgment message, sends an update message to the new host to indicate the new sharers that requested B while it was in transit. The directory state entry of B is consecutively updated.

3 Microarchitecture and Hardware Cost

The ACM mechanism is a completely hardware-oriented scheme. Figure 5 depicts its microarchitecture design within a CMP tile. The added structures are shaded. The ACM engine incorporates the hardware implementation of the ACM algorithm. It further updates the use counters and pattern vectors augmented to cache blocks at the time they are accessed. Note that only one adder per tile is needed to increment the use counters. If the number of accesses reaches the specified migration frequency level maintained by the ACM engine, the pattern vector of the block is read and the optimal host is computed. The ACM engine logic is simple as required by the ACM mechanism. The area and power budget are expected to be modest.

The performance improvement of the ACM mechanism comes at small storage overhead to the on-chip cache hierarchy. We assume 41 bit physical address and 64 byte cache block size. Nine bits of the physical address are used by the nominal shared scheme to index the L2 cache bank within a tile beside six bits for the offset. Four more bits are used for the HS to select the home tile. The tag width is accordingly 22 bits. Each MT table stores tags using a format similar to the tags in the corresponding L2 cache bank. However, for each entry in the MT table, additional 16 bits are required for the directory vector, 1 bit to specify whether the tag is remote or local, and 4 bits to indicate the ID of the tile currently hosting the block. Additionally, augmented to each cache block, are 4 bits to store the use counter (assuming that the maximum allowed migration frequency is 16) and 16 bits for the pattern vector. The table in Figure 6 breaks down ACM's storage requirement for our cache configuration. The MT table has been reduced to quarter its size with a tradeoff of giving up some of the performance gain produced by the ACM mechanism as demonstrated in Section 4. Moreover, the peripheral circuitry and interconnects are not taken into consideration when measuring the percentage increase of the on-chip cache capacity thus the result shown in the table in Figure 6 is in reality much smaller.

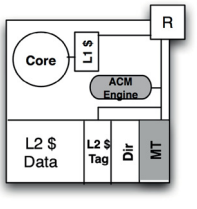


Fig. 5. Michroarchitecture for the ACM mechanism (Figure not to scale)

Component	Bits Per Entry	K Entries	K Bytes Per Tile
TAG	22	2	5.5
Directory Vector	16	2	4
Remote/Local	1	2	0.2
ID	4	2	1
Pattern Vector	16	8	16
Use Counter	4	8	4
Total KBytes			30.7
% Increase of On-Chip Cache Capacity			4.8%

Fig. 6. ACM Storage Overhead

4 Quantitative Evaluation

In this section, we evaluate the ACM mechanism and alternative cache designs. We compare ACM with the non-uniform shared cache scheme (S), as a baseline, and the victim replication (VR) proposal [25] that we consider a traditional design on this direction. The rationale behind such a comparison is to demonstrate that migration, if done in an adaptive-controlled fashion, generates favorable results compared to replication. Replication has been considered useful for tackling the NUCA problem [3, 8]. Migration, on the other hand, hasn't been proved to be highly effective in the context of CMPs [4, 24]. By showing that ACM outperforms the nominal shared cache scheme and one of the conventional replication designs, we thereby demonstrate that migration is in fact an effective technique for managing L2 caches in CMPs. Though there has been more recent work than VR on replication [3, 6, 8], our objective is not to compare against all these, but rather to expose the potential of migration and to elide the fallacy regarding such a technique in the CMP domain.

4.1 Experimental Methodology

Our evaluation employs a detailed full system simulator built on Simics 3.0.29 [2]. We simulate a 16-way tiled CMP architecture organized as a 4×4 2D mesh grid and runs under the Solaris 10 OS. Programs are compiled for an UltraSPARC-III Cu processor. Each core runs at 1.4 GHz, uses in-order issue, and has a 16KB I/D L1 cache and a 512KB L2 cache with the LRU replacement policy. The aggregate L2 cache is consequently 8MB for the 16-tiled CMP model. Each L1 cache is 4-way set associative with 1-cycle access time and 64 byte line. Each L2 cache is 16-way set associative with 6-cycle access time and 64 byte line. A 5-cycle latency per hop, based on a recent processor from Intel [22], is incurred when a datum traverses through the mesh network including both, a 3-cycle switch [9, 24, 25] and a 2-cycle link latencies. The 4-GB off-chip main memory latency is set to 300 cycles.

ACM, S, and VR are studied using a mixture of single-threaded, multithreaded, and multiprogramming workloads. For multithreaded workloads, we use the commercial benchmark SPECjbb, and four other shared memory benchmarks from the SPLASH2

Table 1. Benchmark programs

NAME	INPUT
SPECjbb	Java HotSpot (TM) server VM v 1.5, 4 warehouses
lu	1024×1024 matrix (16 threads)
ocean	514×514 grid (16 threads)
radix	2M integers (16 threads)
barnes	16K particles (16 threads)
parser	reference
art	reference
equake	reference
mcf	reference
ammp	reference
vortex	reference
MIX1	reference for all (vortex, ammp, mcf, and equake)
MIX2	reference for all (art, equake, parser, mcf)

suite [23] (Ocean, Barnes, LU, Radix). For single-threaded workloads we use six programs from SPEC2K [1], three integers (vortex, parser, mcf) and three floating-points (art, equake, ammp) with the reference data sets. These benchmarks were chosen because they demonstrate different access patterns and different working set sizes [1, 10]. Two multiprogramming workloads, MIX1 (vortex, ammp, mcf, equake) and MIX2 (art, equake, parser, mcf), are constructed from the selected 6 SPEC2K programs. Initialization phases of applications are skipped using magic breakpoints from Simics. For the single-threaded and multiprogramming workloads, a detailed simulation is run for each benchmark until at least one core completes 1 billion instructions. Table 1 summarizes all the simulated benchmarks. Last but not least, we fix the migration frequency level to 10 throughout the simulation.

4.2 Simulation Results

In this subsection we report our simulation results and analyze the effectiveness of ACM over S and VR schemes. The aim is to study the efficiency of the ACM mechanism in the presence of *no-sharing*, *little sharing*, and *sharing* of cache blocks. These cases are essentially offered by the single-threaded, multiprogramming, and multithreading workloads, respectively. The main target of the ACM mechanism is to tackle the non-uniformity in access latency that the shared scheme exposes (NUCA problem). Accordingly, the primary evaluation metric that we adopt is the average L2 access latency (AAL) experienced by an access to the L2 cache from any core on the tiled CMP. Upon an L1 miss, an access to L2 can be defined in terms of the congestion delay, the number of network hops traversed to satisfy the request, and the L2 bank access time. Three scenarios may occur thereupon. First, the request may hit in its local L2 bank, and we assume that this incurs only 6 cycles. Second, It may miss locally but hit remotely, thus incurring an access latency that varies depending on the network congestion and the number of network hops. Third, it may miss on chip and consequently reach the main memory to satisfy the request. Thus AAL combines both, the access to L2 (locally or remotely) and the L2 miss rate and penalty. Clearly, an improvement in the AAL metric translates to an improvement in the overall system performance. The average memory access cycles spent in L1, L2, and main memory serving 1K instructions is furthermore shown to give an overall performance picture. The L2 miss rate is also demonstrated to assure the claim that it is maintained by the ACM mechanism and because of its importance to the VR strategy. We also report the message-hops per 1k instructions.

Comparing Schemes, Single-threaded and Multiprogramming Workloads: Multiprogramming workloads tend to have very little sharing among the different threads [6, 24]. Single-threaded benchmarks represent the *no-sharing* case. VR is very appealing in this situation because it can automatically attract data blocks to the only tile running the thread, thus supposedly reducing access latency by decreasing inter-tile accesses from replica hits. However, this may make the tile running the thread experience some high capacity demand. This may result in poor utilization of the on-chip cache capacity. If the scheme fails to offset the increased miss rate then this could lead to performance degradation. This intuition is confirmed by the results shown in Figure 7. The L2 miss rates of all the single-threaded benchmarks shown for VR are all much larger

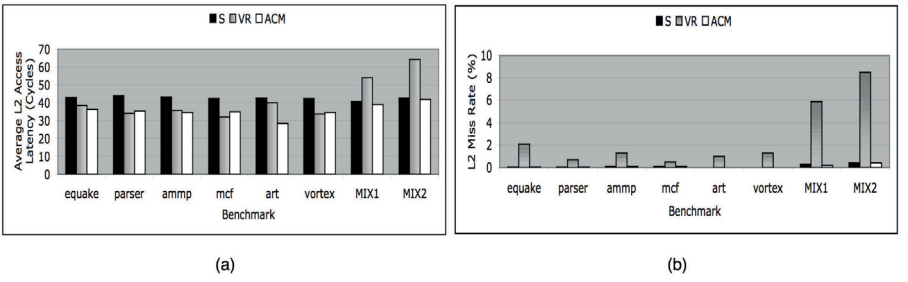


Fig. 7. Single-threaded and Multiprogramming Results (S = Shared, VR = Victim Replication)

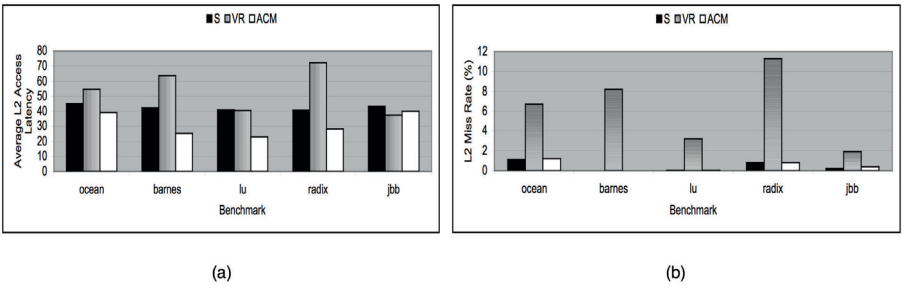


Fig. 8. Multithreaded Results (S = Shared, VR = Victim Replication)

than that for S. However, across all the workloads VR successfully offsets the miss rate from fast replica hits. Contrary to that, VR fails to offset the increase in L2 miss rate for the multiprogramming workloads. Clearly, the SPEC2k applications have small working sets that more or less fit in L1 and L2 caches as they expose negligible L2 miss rates as is shown in the figure for the S scheme. The L2 miss rate for the 6 single-threaded benchmarks is on average 0.04%. As the memory footprint of the benchmark decreases, the space made available to replicas increases and accordingly more performance improvement can be achieved. For the multiprogramming workloads, 4 benchmarks are now sharing the L2 cache space. Hence, the memory footprint of the workload has been increased. The L2 miss rate for MIX1 and MIX2 is now 0.3% on average, or 8.7x more than that of the single-threaded workloads. VR failed to offset this increase and produced 41.8% and 46.0% AAL degradation over S and ACM schemes respectively.

Contrary to that, ACM still offers this automatic data attraction functionality suggested by the VR scheme but in a very controlled fashion that it can efficiently customize allocation of on-chip capacity via the swapping with LRU policy as is discussed in Section 2. Consequently, it successfully generated AALs that are on average 20.5% and 3.7% better than S and VR respectively for the single-threaded workloads, and 2.8% and 31.3% better than S and VR respectively for the multiprogramming ones. VR performs better than ACM only for the benchmarks vortex, parser, and mcf. It has been observed that 81%, 50%, and 57% of the cache blocks of the vortex, parser, and mcf benchmarks respectively are accessed for less than 10 times (the specified migration frequency level). As a result, for all these cache blocks, the ACM mechanism didn't

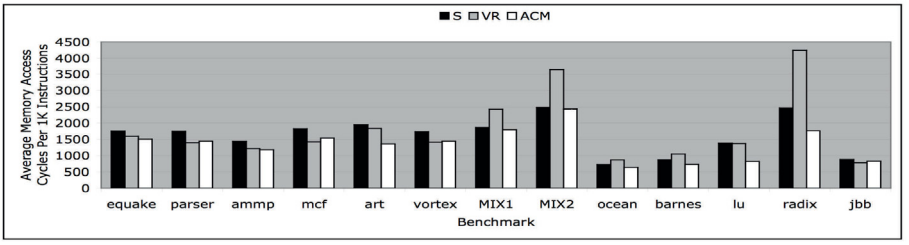


Fig. 9. Average Memory Access Cycles Per 1K Instructions Results (S = Shared, VR = Victim Replication)

even attempt to migrate them to better hosts so as to minimize the L2 access latency. This is because we fixed the migration frequency level throughout simulations. Migration to any cache block is triggered upon being accessed for the number of times that the migration frequency level specifies. For that reason, ACM is not exhibiting its full ability to exploit the optimum performance though it still on average greatly surpasses both of the schemes, S and VR. With an adaptive tunable migration frequency level, the ACM mechanism would hit its optimum and consequently provide larger performance improvements. This is to be explored in future research work.

As clearly shown in Figure 7, ACM maintains the L2 miss rates of S for all the simulated single-threaded and multiprogramming benchmarks. Moreover, it optimizes some of them because of the swapping with the LRU policy and generates on average 2x and 1.5x reductions in L2 miss rates over S for the single-threaded and the multiprogramming benchmarks respectively. Finally, Figure 9 shows the average memory access cycles per 1K instructions experienced by all the simulated benchmarks. VR performs on average 15.1% better than S and 38.4% worse than S for the single-threaded and the multiprogramming benchmarks respectively. ACM, on the other hand, performs on average 18.6% and 2.6% better than S, and 3.4% and 29.4% better than VR for the single-threaded and the multiprogramming benchmarks respectively.

Comparing Schemes, Multithreaded Workloads: The multithreading workloads expose different degrees of sharing among threads and accordingly allow us to study the efficiency of the ACM mechanism with such a case. Figure 8 depicts AALs and the L2 miss rates of the multithreading workloads compared to S and VR. ACM exhibits AALs that are on average 27.0% and 37.1% better than S and VR respectively. VR reveals 26.7% worse AAL than S for all the simulated benchmarks. This is due to the fact that VR has a static replication policy that depends on the blocks' sharing behaviors. An increase in the degree of sharing suggests that the capacity occupied by replicas could increase significantly leading to a decrease in the effective L2 cache size. As such, if replicas displace too much of the L2 cache capacity, the L2 miss rate could increase considerably, degrading thereby the average L2 access latency. This was clearly illuminated by the behaviors of the Ocean, Barnes, and Radix benchmarks where reduction in latencies via replica hits failed to offset the excessive latencies deduced by the increased miss rate. Barnes for instance utilizes a tree data structure that exhibits a sharing degree of 71% [4] and accordingly incurs a significant increase in capacity pressure when VR

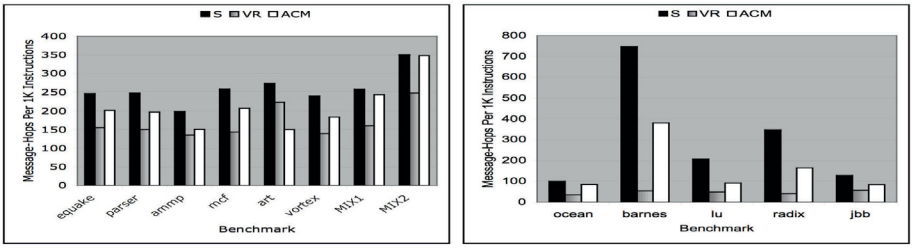


Fig. 10. On-Chip Network Traffic Comparison (a) Single-threaded Workloads (b) Multithreaded Workloads

is used. Note that such an inferred VR behavior is more elucidated in this work than in the original evaluation [25] as the L2 cache has been downsized to half. VR was successful in offsetting the impact of increased offchip accesses for the LU and JBB workloads.

On the other hand, ACM, a pure migration technique, maintains the exclusiveness of cache blocks on chip and consequently preserves the L2 miss rates of S for the three benchmarks Barnes, Lu, and Radix. Only Ocean and JBB reveal a small increase in the L2 miss rate for ACM over S. This is because when some block, B1, is to be migrated to a new host, H, no valid entry for the LRU block, B2, that is to be swapped with B1, is found in the MT table of H, and accordingly discarded as planned by the swapping with the LRU policy. That discarded block, B2, can be requested again by some other threads. VR only performs better than ACM for the JBB benchmark. The reason is the fixed migration frequency level that we assume throughout the simulation process. We ran JBB with doubling and tripling the migration frequency and respectively obtained 3.7% and 6.7% more AAL improvements over the base run with a migration frequency of 10. Lastly, Figure 9 shows the average memory access cycles per 1K instructions. For the multithreading benchmarks, VR performs on average 19.6% worse than S. ACM, in contrary, performs on average 20.7% and 29.7% better than S and VR respectively.

On-Chip Network Traffic: A supplementary advantage of the ACM mechanism is the reduced on-chip network traffic that it offers. Figure 10 depicts the number of message-hops per 1k instructions that the three schemes, S, VR, and ACM exhibit for the single-threaded, multiprogramming, and multithreaded workloads. The ACM scheme offers 25.3%, 3.0%, and 41.6% on-chip network traffic reduction over S for the single-threaded, multiprogramming, and multithreaded workloads respectively. The VR scheme, on the other hand, offers 35.6%, 33.6%, and 75.7% on-chip network traffic reduction over S for the three workloads respectively. Consequently, VR offers more on-chip network reduction over S than what ACM does because it decreases more inter-tile accesses from replica hits. Though the ACM mechanism bears some resemblance to the VR strategy for the single-threaded workloads as discussed in Section 2, but with a fixed migration frequency level of 10, a tile waits for 10 accesses to the block to attract it to its local L2 bank. Therefore, it incurs more inter-tile accesses compared to VR that tries to attract the block to its local L2 bank immediately after being evicted

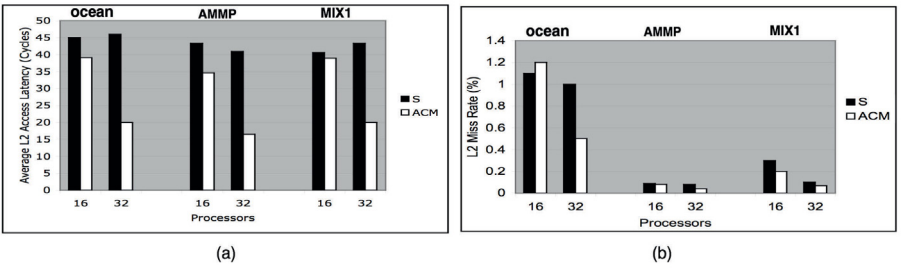


Fig. 11. Results for CMP Systems with 16 and 32 Processors (a) Average L2 Access Latencies (b) L2 Miss Rate

from L1. If VR fails to replicate cache blocks while ACM succeeded to attract blocks to its local L2 bank, ACM will surpass VR. The Art benchmark is the only case that exhibits such a situation. Finally, we studied the increase in network congestion for ACM over S without employing the cache the cache-tags policy. We found on average an increase of 17.6%, 4.1%, and 1% over S for the single-threaded, multiprogramming, and multithreaded workloads respectively. Clearly, this demonstrates the decisiveness and usefulness of such a policy when applying block migration in CMPs.

ACM Adaptability to Futuristic CMP Models: As the number of tiles on a CMP platform increases, the NUCA problem exacerbates. In order for any cache management scheme to be useful in tackling the NUCA problem, it should demonstrate adaptability to upcoming futuristic CMP models. ACM is very expedient in this direction as it always selects a host for a cache block that minimizes the total L2 access latency for all the sharers of that block independent of the underlying CMP size. Thus more exposure to the NUCA problem translates effectively to a larger benefit from the ACM scheme. We show such a pro of ACM via extending our CMP model to 32 tiles and running simulations for three selected benchmarks. Each tile still maintains, as with the 16-tiled CMP model, a 16KB I/D L1 cache and a 512KB L2 cache bank. The three benchmarks that have been chosen to conduct the study are, Ocean, ammp, and MIX1, from the multithreaded, single-threaded, and multiprogramming workloads respectively. These benchmarks revealed the highest L2 miss rates among the others in their sets; hence, selected.

Figure 11 depicts the AALs and the L2 miss rates of the selected benchmarks. For 16 tiles, ACM shows AAL improvements of 13.1%, 20.0%, and 1.7% for Ocean, ammp, and MIX1 over S respectively. However, for 32 tiles, ACM shows AAL improvements of 56.5%, 59.6%, and 53.8% over S respectively. As a result, ACM exhibits on average 11.6% and 56.6% AAL improvements over S for the 16-tiles and 32-tiles models respectively. The 32-tiles CMP model produced latency results that are 4.8x times better than those generated by the 16-tiles one for the simulated benchmarks.

Sensitivity and Stability Studies: So far we have assumed for simplicity that the size and associativity of the MT tables are identical to that of the L2 caches. There is nothing, in fact, that prevents this data structure from being of a smaller size or associativity. To study the sensitivity of the ACM mechanism to this component, we ran simulations

for the three benchmarks, Ocean, ammp, and MIX1 with MT table sizes reduced to *half* (50%) and *quarter* (25%) the size of the *base* cache, and with a 16-way set associativity. With half and quarter configurations, we got AAL increases of 5.9% and 11.3% respectively over the base one, but still improvements of 7.6% and 2.9% respectively over S. The highest contribution for the AAL increases was from the ammp benchmark. The ammp benchmark shows alone 19.9% AAL increase, averaged for both the half and quarter configurations, over the base, though still 6.1% better than S. It was observed that 60.7% of the cache blocks in ammp are accessed at least 10 times (the specified migration frequency level) before getting evicted from L2, consequently triggering migrations. To decrease the pressure on the MT table, we ran simulations with migration frequency of 20 rather than 10. Compared to the 19.9% AAL increase, we obtained only 12.3% increase, averaged for both the half and quarter configurations, over the base one and consequently 10.1% on average better than S.

Finally, and to demonstrate the stability of the ACM scheme to different cache sizes, we simulated the LU benchmark on our 16-tiled CMP model with the L2 cache being reduced to *half* its size for the three different schemes: S, VR, and ACM. VR failed to demonstrate stability and showed AAL degradation of 37.8% over S, while ACM maintained AAL improvement of 39.7% over S. This is because the ACM mechanism maintains the exclusiveness of cache blocks on chip, while VR demands more capacity to store replicas. Clearly, this reveals the effectiveness of the migration technique in the CMP domain, and particularly that of the proposed ACM mechanism.

5 Related Work

Migrating data to improve memory access latency has been extensively studied in the context of distributed shared-memory multiprocessors [5, 11, 12, 19]. Kim et al. [16] proposed D-NUCA as a mechanism that allows important data to migrate towards the processor within the same level of cache in the context of uniprocessor. Beckmann and Wood [4] examined block migration in CMPs and suggested CMP-DNUCA similar to the original D-NUCA proposal. They employ a gradual migration policy and move blocks along 6 bankcluster chain (the cache banks are physically separated into 16 different bankclusters). However, high degree of cache block sharing complicates the policy and blocks tend to congregate at the center of the cache away from all the processors.

A recent study [17], undertaken independently, proposed an efficient migration scheme to address the NUCA problem via modeling it in the L2 space as a two-dimensional post office placement problem. The scheme determines a suitable location for each data block at any given time during execution. Though the proposed design bears resemblance to ACM in that it dynamically finds a proper L2 location for each cache line after tracking its cache pattern, it doesn't discuss the mechanism with which migrating cache blocks is efficiently done.

Zhang and Asanović [24] proposed data migration in the context of tiled CMP. They allow replication and employ a migration strategy similar to the first touch policy presented in Section 1, but with certain conditions to move cache blocks to the requester tiles. Lastly, many proposals in the literature advocate data replication to manage the

non-uniformity in latency exposed by the nominal shared L2 cache design [3, 8, 25]. A recent study [3] demonstrated that blind replication is dangerous because the capacity occupied by replicas could increase significantly. The study proposed a hardware-based mechanism that dynamically monitors workload behavior to control replication.

6 Conclusions and Future Work

Managing L2 caches in chip multiprocessors is essential to fuel its performance growth. This paper studied a strategy to manage non-uniform shared caches in CMP by dynamically migrating cache blocks to optimal locations that provide the minimal L2 access latency. The proposed mechanism optimizes the L2 miss rate via maintaining the uniqueness of cache blocks on chip. Besides, Cache the Cache-tags policy has been proposed to effectively simplify the process of locating migratory blocks. Simulation results demonstrated the robustness, scalability, and stability of ACM. Unlike previously studied migration strategies in CMP literature, the proposed mechanism revealed and confirmed the usefulness of data migration in chip multiprocessors.

The strategy that we proposed to locate optimal hosts for cache blocks is simple and assumes that if a block has been accessed by a certain core in the past, then it is likely to be accessed by the same core in the future. Chishti et al. [8] observed that many blocks brought to the cache are not reused in some workloads. Thus, the proposed hardware host predictor can be easily improved by introducing more weights for the cores that accessed a cache block often.

Finally, we fixed throughout our simulation process the migration frequency level to 10. As is shown, this doesn't exhibit the full capability of the ACM mechanism. An adaptive tunable migration frequency level would allow the ACM mechanism to hit its optimum. Having established the effectiveness of the main idea of ACM, improving performance through an adaptive algorithm to dynamically tune the migration frequency level is the obvious next step.

References

1. Standard performance evaluation corporation, `http://www.specbench.org`
2. Virtutech, A.B.: Simics full system simulator, `http://www.simics.com/`
3. Beckmann, B.M., Marty, M.R., Wood, D.A.: Asr: Adaptive selective replication for cmp caches. In: MICRO (December 2006)
4. Beckmann, B.M., Wood, D.A.: Managing wire delay in large chip-multiprocessor caches. In: MICRO (December 2004)
5. Chandra, R., Devine, S., Verghese, B., Gupta, A., Rosenblum, M.: Scheduling and page migration for multiprocessor compute servers. In: ASPLOS (October 1994)
6. Chang, J., Sohi, G.S.: Cooperative caching for chip multiprocessors. In: ISCA (June 2006)
7. Chishti, A., Powell, M.D., Vijaykumar, T.N.: Distance associativity for high-performance energy-efficient non-uniform cache architectures. In: MICRO (December 2003)
8. Chishti, Z., Powell, M.D., Vijaykumar, T.N.: Optimizing replication, communication, and capacity allocation in cmps. In: ISCA (June 2005)
9. Cho, S., Jin, L.: Managing distributed shared l2 caches through os-level page allocation. In: MICRO (December 2006)

10. Dybdahl, H., Stenstrom, P.: An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In: HPCA (February 2007)
11. Falsafi, B., Wood, D.A.: Reactive numa: A design for unifying s-coma and cc-numa. In: ISCA (June 1997)
12. Hagersten, E., Landin, A., Haridi, S.: Ddm-a cache-only memory architecture. IEEE Computer (September 1992)
13. Held, J., Bautista, J., Koehl, S.: From a few cores to many: A tera-scale computing research overview. White Paper. Research at Intel. (January 2006)
14. Kim, C., Huh, J., Shafi, H., Zhang, L., Burger, D., Keckler, S.W.: A nuca substrate for flexible cmp cache sharing. In: ICS (June 2005)
15. Johnson, T., Nawathe, U.: An 8-core, 64-thread, 64-bit power efficient sparc soc. In: IEEE ISSCC (February 2007)
16. Kim, C., Burger, D., Keckler, S.W.: An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In: ASPLOS (October 2002)
17. Li, F., Kandemir, M., Irwin, M.J.: Implementation and evaluation of a migration-based nuca design for chip multiprocessors. In: ACM SIGMETRICS (June 2008)
18. Marty, M.R., Hill, M.D.: Virtual hierarchies to support server consolidation. In: ISCA (June 2007)
19. Mizrahi, H.E., Baer, J.L., Lazowska, E.D., Zahorjan, J.: Introducing memory into the switch elements of multiprocessor interconnection networks. In: ISCA (1989)
20. Mullins, R., West, A., Moore, S.: Low-latency virtual-channel routers for on-chip networks. In: ISCA (June 2004)
21. Sinharoy, B., Kalla, R.N., Tendler, J.M., Eickemeyer, R.J., Joyner, J.B.: Power5 system microarchitecture. IBM J. Res. & Dev. (July 2005)
22. Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Iyer, P., Singh, A., Jacob, T., Jain, S., Venkataraman, S., Hoskote, Y., Borkar, N.: An 80-tile 1.28tflops network-on-chip in 65nm cmos. In: ISSCC, New York (February 2007)
23. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The splash-2 programs: Characterization and methodological considerations. In: ISCA (July 1995)
24. Zhang, M., Asanović, K.: Victim migration: Dynamically adapting between private and shared cmp caches. Technical Report TR-2005-064, Computer Science and Artificial Intelligence Labratory. MIT (October 2005)
25. Zhang, M., Asanović, K.: Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In: ISCA, New York (2005)

In-Network Caching for Chip Multiprocessors*

Aditya Yanamandra, Mary Jane Irwin, Vijaykrishnan Narayanan, Mahmut Kandemir, and Sri Hari Krishna Narayanan

Department of Computer Science and Engineering,
The Pennsylvania State University
{yanamand,mji,vijay,kandemir,snarayan}@cse.psu.edu

Abstract. Effective management of data is critical to the performance of emerging multi-core architectures. Our analysis of applications from SpecOMP reveal that a small fraction of shared addresses correspond to a large portion of accesses. Utilizing this observation, we propose a technique that augments a router in a on-chip network with a small data store to reduce the memory access latency of the shared data. In the proposed technique, shared data from read response packets that pass through the router are cached in its data store to reduce number of hops required to service future read requests. Our limit study reveals that such caching has the potential to reduce memory access latency on an average by 27%. Further, two practical caching strategies are shown to reduce memory access latency by 14% and 17% respectively with a data store of just four entries at 2.5% area overhead.

1 Introduction

Effective management of shared data is critical to the performance of emerging multi-core architectures. Consequently, the use of shared L2 cache architectures has emerged as a popular trend to facilitate data sharing. Maintaining cache coherence is a critical need in such shared structures and is commonly enforced using either snoop-based or directory based protocols. In the broadcast-based snoop protocol, a requesting node broadcasts a snoop request to all nodes to find whether they currently cache the data or not. In contrast, the directory-based protocol is more scalable with number of cores as they eliminate the broadcast with directed messages according to the directory information. The directory-based protocol manages which node caches data in a directory corresponding to the home node at the requested address. Whenever, a data is requested by a processor, the shared directory entry is used to identify the state of requested data as well as the current location of the shared data. Consequently, the data request involves three steps: (1) Communication from the requesting node to the home node of the directory entry; (2) In case, a shared copy exists, a message is sent from the home node to the location of remote node caching the data. In case, no copy exists, the next level cache or memory is accessed and data is returned to requesting node skipping step 3; (3) a communication packet forwarding the data from the remote node with shared data to the requesting node. However, this protocol of the shared directory suffers from

* This research is supported in part by NSF grants 0702617, 0811687, 0720645, 0720749, 0702519, 0444345 and a grant from GSRC.

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 373–388, 2009.

the latency overhead of having to send the data request in the order of the requesting node, the home node and the caching node.

Due to the communication-centric nature of such shared data accesses, the underlying communication fabric and protocols play an important role in determining the data access latency. There have been several efforts at reducing the overheads of directory-based protocols and on the design of low-latency communication fabrics. In this work, we focus on the design of an efficient packet-based on chip communication fabric to reduce the number of hops required in servicing a data request. Particularly, we focus on caching data along the routers of the multi-hop communication fabric to facilitate faster response to data accesses. Most closely related to our effort is the effort to migrate the entire data cache on to the network layer by Mirzahi et al. [14]. In order to circumvent the resource challenges of migrating large amount of data into the network router in an on-chip network, Eisley et al. [6]. proposed caching only the coherence protocol information rather than the actual data. Our work identifies that caching a small fraction of shared data within the router in addition to caching protocol information can provide significant performance gains.

The key contributions of this paper are

- Analyzing the memory access patterns of a set of applications from SpecOMP [1] suite and identifying that a small number of shared addresses dominate the accesses to memory systems and reducing access latencies to these shared locations has a significant impact on overall memory latency.
- Exploration of three different in-network data caching strategies to reduce overall latency that indicates that memory access latency can be reduced by upto 38% and on average by 27% across the applications.
- Design of the router micro-architecture to support in-network data caching to demonstrate the feasibility of incorporating them in on-chip networks.

The rest of this paper is organized as follows. The next section analyzes the memory access behavior of the SpecOMP [1] benchmark suite and provides an overview of In-network cache coherence. Section 3 describes the modifications to the protocol and the microarchitectural changes in the router architecture required to enable and implement In-network caching. Section 4 explores our proposed In-network caching policies. Section 5 details the experimental setup for this work. Section 6 provides quantitative evaluation of the in-network caching schemes along with the area overhead of the design. Section 7 discusses the related work. We conclude this work in Section 8.

2 Motivation and Background

Access to shared data is one of the critical factors that impact the performance of parallel algorithms. To understand this better we profile the SpecOMP benchmark for statistics on the number of addresses and the number of accesses to them. We classify these statistics as private and shared. We define a private address as one which is accessed by only one processor and is thus private to a single processor. A shared address is one which is accessed by at least two processors. Figure 1 shows the distribution of private and shared addresses for the profiled applications. The results indicate that the

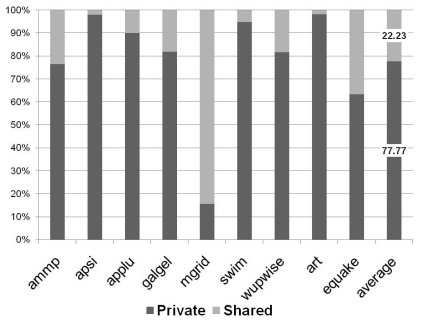


Fig. 1. Ratio of the number of private to shared addresses

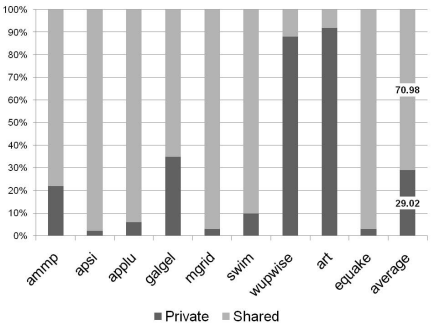


Fig. 2. Ratio of the accesses to private and shared addresses

number of distinct shared addresses accessed is a small fraction of all addresses accessed. In addition to this observation, we noticed that the number of accesses to the shared addresses accounted for a large fraction (71% on average) of all memory accesses (See Figure 2). These two observations indicate that reducing the access latency to a small fraction of shared address locations may have a significant impact of overall latency. This motivates our effort to cache the shared data in a small data store of on-chip routers.

2.1 In Network Cache Coherence

Our technique is built on top of the in-network cache coherence [6] work that is reduces the hop count for data accesses by caching coherence information within the routers. Hence, we provide a brief overview of this protocol (readers are referred to [6] for details.) The key to the in-network cache coherence work is a virtual tree of links that connects all sharers of the data with the home node. The location of the home node is determined statically based on the address of the data as in the case of traditional directory-based protocols [1].

The node that caches the shared data first serves as the root of the virtual tree and links are maintained at each router in the path between the home node and the sharing nodes. These virtual trees enable new read requests to identify the location of the shared data from the coherence information on the routers instead of having to first traverse to the home node.

Read Operation

When a node (R1) issues a read request, the packet is transmitted towards the home node (H) as shown in Figure 3(a). There two possible scenarios in handling this request. In the first case, the data is not cached in any other node and no outgoing virtual link exists from the home node. In this case, H loads the data from the secondary cache or memory and responds to the requester with data. When the response packet traverses from the home node towards the requester (R1), the virtual links in each intermediate router are set to point towards the requesting node. Figure 3(b) illustrates the virtual tree formed as a result of such an occurrence.

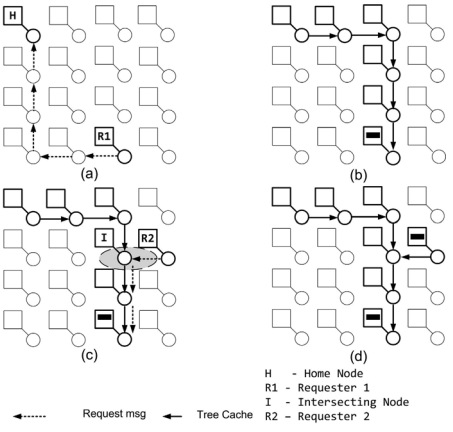


Fig. 3. Read operations for the basic In Network Cache coherence (MSI) protocol. *(a) A read request sent to the home node (b) A read reply message forms the virtual tree rooted at the requester (c) A second read request is rerouted towards the root after hitting the tree en route to the home node (d) The second requester obtains the data and becomes part of the virtual tree*.

In the second case, the requested data is already cached by another node. In Figure 3(c) when R2 is requesting data for an address, R1 has the latest copy of the data and there is tree with the values. (A request from R2 in Figure 3(c)). There are two sub-cases based on whether the path between the requester and the home node (H) intersects with an already formed virtual tree. In the first sub-case, when the request does not intersect with a virtual tree, the request reaches the home node. This is followed by a traversal through the virtual links towards the root node. At each hop along the virtual tree, the local node checks for a cached copy of the shared data. If the line is present in any of the intermediate nodes on the path, the request is satisfied and a response packet to the requester is formed. If not, the response packet originates from the root node. The response packet modifies the virtual tree on its way to the requester. Figure 3(d) depicts the augmented virtual tree. The second sub-case is similar to the first sub-case, except that the request is routed towards the root of the virtual tree when the request intersects with the virtual tree link at the node I. Consequently, it reduces the traversal all the way to the home node.

Write Operation

When a node requests for the write operation, a packet is transmitted towards the home node(H) for the address . In case it intersects a tree for the same address in some router along the way, that tree needs to be torn down before the current request is granted. In Figure 4(a) the write request from R2 intersects a tree on its way to the H. Then, the current node issues teardown messages for that address and sends it in all directions of the tree. Once the teardown message reaches a leaf in the directed tree, it invalidates the line for the entry in it's tree cache. Further, it sends an acknowledgement up the tree, as shown in Figure 4(b). As a result of the mechanism, all the acknowledgements head towards H as shown in Figure 4(c). Once H becomes the leaf of the tree (every path out

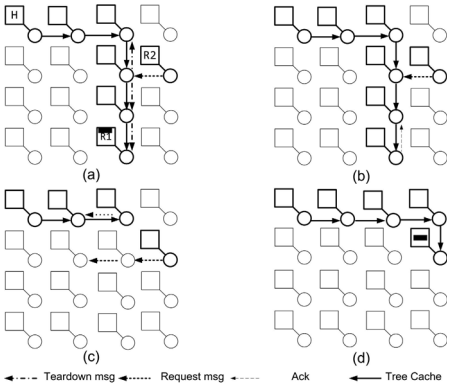


Fig. 4. Write operations for the basic In-network Cache coherence (MSI) protocol *Write operations for the In Network Cache coherence (MSI) protocol. (a) A read request sent to the home node intersects an existing virtual tree causing teardown message propagation (b) Acknowledge messages from the leaf node (c) Acknowledge message causing teardown and the write request message in flight (d) The new virtual tree with the requester as the root.*

of it is invalidated) it then grants the request for R2. This results in forming a tree to R2 as shown in 4(d).

3 In-Network Data Caching

This section describes the necessary augmentations to the in-network cache coherence protocol to support the proposed data caching technique (Figure 5). We also show the micro-architectural modifications required to support the new functionalities along with a brief description of the implementation.

3.1 Protocol Modifications

Reads: Changes occur in the behavior of both the read reply message and the read request message. The data from a read reply message passing through an intermediate node towards the requester, can potentially be cached in the router's data store depending on several factors, including availability of free lines and the caching policy. Any read request message will intersect the virtual tree for the cache line either at the home node or at an intermediate node. In the basic version of the protocol, along the path of a tree, each node is checked to see if it has a local copy of the data. In our scheme, along with a check at the node, a *simultaneous check is made in the data store of the router*. If no copy exists, the read request message is routed towards the root as usual. Once a copy of the data is found either in the intermediate node's cache or the data store of its router, a read reply message is generated. A hit in the data store can reduce the number of hops needed to put the reply in the network and the hops the reply needs to take to its destination.

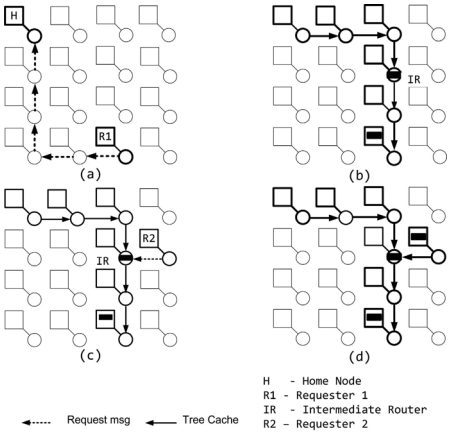


Fig. 5. Read operations for the proposed *In Network Caching (MSI) protocol. (a) A read request sent to the home node (b) A read reply message forms the virtual tree rooted at the requester; The cache line is stored at an intermediate router (c) A second read request intersects the virtual tree at node with cache line in the router data store (d) The second requester obtains the data from the data store and becomes part of the virtual tree.*

Writes: When a write request intersects the virtual tree of the cache line for which the write has been requested, it causes teardown messages to propagate towards the leaf nodes in the tree. At each intermediate node, in addition to the basic actions described in Section 2.1, any copy of the cache line in the data store of the router is marked as invalid. The write latency to the caches is not directly changed by our protocol. Any change in the write latency will be from the new traffic in the network which can arise from the altered read-response traffic.

Teardown: As soon as a teardown message is seen in the router, a check is made to see if the teardown address is a match in the router's cache. In case there is a hit, the address is invalidated immediately. The rest of the protocol remains the same as in the In-Network Cache Coherence approach.

3.2 Router Microarchitecture

The proposed router architecture (shown in Figure 6(a)) has ports to access its four neighbors and it's local processing element (PE). The proposed router pipeline that supports the modified protocol is shown in the Figure 6(c). It consists of the Route Compute (RC) stage, a Virtual Channel Allocation (VA) stage, a Switch Allocation (SA) stage, a Switch Traversal and a Link Traversal stage. When a packet enters the pipeline, the RC unit uses X-Y routing to route the packet.

The traditional router pipeline was enhanced with a *Tree Cache (TC)* stage to enable In Network Cache Coherence [6]. The Tree Cache stage maintains the virtual links that form the tree. These virtual links are used to redirect requests for data and hence help reduce the hops taken by the request packet before it reaches a copy of the data.

This work adds another unit called the **Router Local Cache** (RLC) to the pipeline to be used for data storing. The structure of the RLC is as shown in Figure 6 (b). The

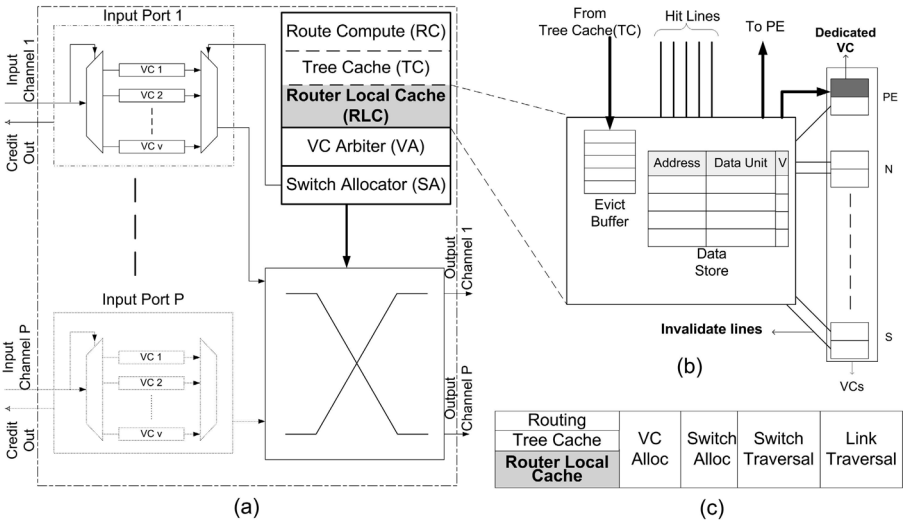


Fig. 6. Proposed router Microarchitecture. *(a) The router consists of P input ports consisting of V virtual channels and P output ports connected to a crossbar. Packets pass through the router pipeline between the virtual channels and the switch. (b) The proposed Router Local Cache stage that manages the data in the Data Store in the router (c) The router pipeline. The Router Local Cache stage occurs in parallel with the packet routing and TC stage.*

main components of the RLC are *Data Store*, *Invalidation lines to the VC* and the *Evict Buffer*. This unit implements policies to store cache lines observed at the router into the data store during a read reply message. It also implements policies that define when the data in the data store is to be invalidated when it observes a write request or a teardown message. Finally, it also recognizes packets containing read requests for cache lines stored in it and responds appropriately.

Data Store. The Data Store is the main component of the RLC. It is a set of entries where each entry contains a cache line address and a copy of the latest value for that address along with a *valid bit* as shown in Figure 6(b). It is a five-ported structure handling upto five packets in a single cycle. The address of the cache line contained in each packet that passes through the RLC is checked for a match with the addresses contained in the Data Store. The Data Store consists of two main structures, the *Hit Lines* and the *Dedicated VC* which are described below.

Hit Lines: Every read reply packet contains a cache line that can be stored in the data store. However, to maintain coherence, the address of the line should either be previously present in the TC or be inserted in the current cycle in the TC before data is cached in the data store. In some scenarios, such as a conflict in the TC which requires special handling, the read reply packet waits in the VC until the one of the TC lines (and hence and entire tree) is invalidated. The data of such packets are not stored as they are not a part of the TC. The communication of whether the address of a packet is hit in the TC is accomplished by using 'hit lines'. Hit lines are wires, one for each port of the

RLC, that are used to check for the presence in the TC of the packet entering through the corresponding port.

Dedicated VC: A hit for a read request packet in the data store creates a response packet for the data from the Data Store's entry for that address. If the original packet that experienced the hit is from the PE, then the reply is *immediately ejected to the PE.* Any contention for the PE port is handled in a manner similar to the Early ejection [10]. For packets from the remaining ports, a *VC is dedicated* in the input buffers for the PE as shown in Figure 6(b). Dedicating a VC ensures that the new packet that is sent out does not have to search for a free VC every time when there is a hit in the Router Cache. This does not adversely effect the packets from the PE as the injection rates for applications are quite low. However, there are two limitations to this method. The number of packets that can be sent out as a response every cycle are now limited by the number of dedicated VCs (one in our case). Note that, out of the four directional physical channel packets entering a router, there can be potentially four hits in the Data Store. Since there is only one VC into which responses go, three of the hits are not responded to from this data store. Notice that, while not responding from the Data Store for all hits that occur decreases the performance benefits that can be acquired from our technique, it does not cause an error in the request procedure. The request packets that are not responded from the data store, are forwarded as usual along the tree towards the root. They can be responded either by the RLC unit of a different router or ultimately from the root as in the base protocol. The second limitation to this method has to do with the dedicated VC being occupied at the time of a hit in the Data Store. In that case, the current RLC can't service the request packet. This again effects the performance gains achievable from our work. In our simulations, we found the occurrence of these cases to low enough not to affect the actual performance gains. Our experimental setup models these two limitations. A hit in the data store for addresses contained in teardown and write request packets results in the data stored in the Data Store entry for that address being invalidated. This is accomplished by resetting the valid bit of the entry.

Invalidation Lines. When a packet enters the router, it is placed in a VC. From the next cycle, this packet arbitrates for the virtual channels of the next router. If this packet is a read request that is responded to from the Data Store, it needs to be restricted from further propagation. This is achieved by invalidating the VC. This invalidation is accomplished using the invalidation lines shown in Figure 6(b). There is one invalidate line for each VC of the router, which can set or reset the valid bit of each of these VCs.

Evict Buffer. The condition that needs to be met to maintain cache coherence is as follows "If a line is valid in the RLC, it should be valid in the TC as well." The evict buffer ensures that this condition is held. If at a given instance, the Data Store contains an entry for an cache line that is present in the TC, any state changes (invalidations) for the lines in the TC need to be reflected in the entries of the Data Store as well. A change in state for a line for a TC can occur in two ways 1) through an external packet or, 2) through internal protocol transitions. In the case of teardown and write request messages, a conservative approach is adopted and the cache line in the Data Store is immediately invalidated. This is accomplished through the appropriate 'valid

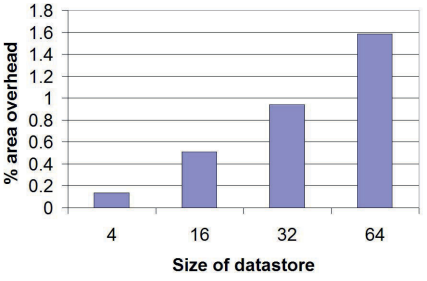


Fig. 7. Area overheads for various sizes of the datastore unit compared to baseline (Figure 6)

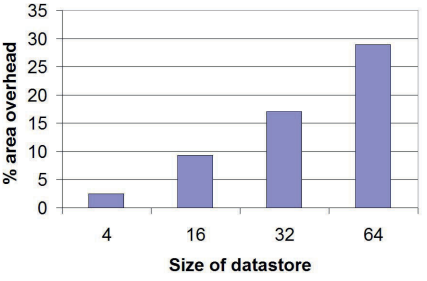


Fig. 8. Area overheads for various sizes of the datastore unit without the TreeCache unit

bit' entry. Internal state changes include evictions of the TC line due to conflicts. A line that is evicted from the TC due to a conflict must be evicted in the next cycle from the data store to maintain coherence. In addition to this, to maintain coherency, we modify eviction policy. This is handled in conjunction with a modified replacement (eviction policy) which ensures that when a line is being evicted from the TC, another packet does not have a hit to the same line. However, in the corner case, all the ways of a TC cache can be a hit to the incoming packets. In such cases, the packet requesting the eviction, backs off for a cycle before trying to evict line again. To invalidate the evicted addresses in the following cycle, all the addresses which are evicted in the TC in the current cycle are collected into an *Evict Buffer* as shown in Figure 6(b). These addresses are invalidated from the RLC in the following cycle. This way we ensure that the correctness condition stated above is satisfied.

Finally, the set of packets that are to be processed in the data store in a given cycle include the potential five new packets and a potential 5 new evictions (all packets from the preceding cycle evict). The Data Store is a five-ported structure and can thus handle only five packets in a given cycle. Not handling any of the packets might lead to a coherence violation in the Data Store. We handle this situation of more packets to process than available ports by simply invalidating the entire Data Store. Our experiments revealed that the occurrence of these events is quite rare due to the low injection rates of applications and thus doesn't affect the performance of our system.

3.3 Implementation

The RLC stage was designed in HDL and was synthesized at the 90nm technology. From this, we saw that the TC stage was still the frequency determining stage for all sizes of data store considered in this work. Thus, our work doesn't reduce the frequency of the baseline router. The increase in area of the router in shown in Figures 7,8. Figure 7 shows the increase in area for different sizes of data store compared to the baseline router. In this instance, the area overhead is less that 2% for our largest sized data store. The TreeCache(TC) stage replaces the directories in the cache subsystem. Thus, we also compare the impact on area for a baseline router without the TC stage. Figure 8 shows the percentage increase in area compared to the baseline router without a TC. Here the area overheads are higher and vary from 2.5% to 29%.

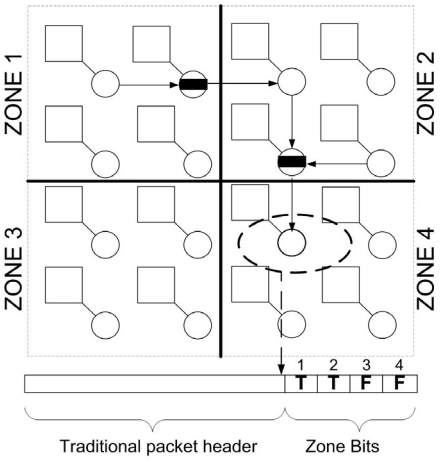


Fig. 9. Zones in a mesh: *The implementation of Zonal Caching using zonal bits. Zonal bits indicate whether a cache line is currently cached within a particular zone. The zone bits of the message show that is it cached in Zones 1 and 2 and not cached in Zones 3 and 4. Hence, it may be cached at the highlighted node.*

4 In Network Caching Policies

In this section, we detail the caching policies explored in this work. When a node is replying to a read request, it forms a 'read reply' message and sends it to the network. A read reply message always follows the tree back to the request node. In case it arrives at a node without a tree, the node is added to the tree structure before forwarding the packet. When a new read reply packet enters the router, a decision needs to be made on saving the data from the packet in the Data Store. We experiment with three methods of making this decision namely Ideal Caching, Independent Caching and Zonal caching.

4.1 Ideal Caching

This scheme demonstrates the best case performance that can be obtained from storing data in the routers. We model ideal caching by simulating an infinite sized data store. Thus, the decision made in every router is to store data from all the 'read reply' packets. Though impractical to be actually implemented, this experiment gives us the upper limit on the amount of savings possible through in-network caching. This also helps us understand the effectiveness of the more practical schemes that follow. In the ideal caching, every node in the tree cache for a shared data has the data as well. Thus, for a read request packet, intersecting a tree for the requested address is equivalent to finding the data.

4.2 Independent Caching

In our next caching policy, we look at an easy to implement caching scheme. In this scheme, the router has a fixed size for the local store. In independent caching, every

router decides what it wants to keep in the additional storage independently. Thus, the decision is made by each router without any global knowledge. Each router looks at the data passing through it and decides whether to cache it or not. This policy is implemented in the following way. The router tries to cache all the data that passes through it. If the router's data store is already full, we replace the least recently used line. However, we do not evict a line unless it's last access has been before a threshold number of cycles (50 cycles)to the current cycle. This is done to prevent evicts from the router's data store frequently. The biggest drawback with this scheme is that it underutilizes the local store capacity. For instance, if a data packet passes in a particular direction, then it has the potential of being cached in every router on the path. However, given the fact that the number of shared addresses is few, this caching scheme is still effective.

4.3 Zonal Caching

The drawback of the independent caching scheme is that it might cache multiple copies of the same data in the network. The best way to use the available capacity of the router's cache is to ensure that there is just a single copy of this data in the network. However, trying to reduce the number of copies also conflicts with the goal of reducing the number of hops to service the request.

Zonal caching aims to compromise between these two conflicting goals. This scheme aims to increase the effective data store capacity over the Independent caching scheme while not restricting the number of copies to one. Multiple copies are allowed in the network as long as they are distant from each other. To realize such a scheme, some global state information is required on where the copies of the data are present in the data stores of the network. Carrying explicit information on the location of the data in the router cache space gives the most information to make such decisions. However, it is not scalable.

As a more scalable approach, we divide the mesh of routers into a grid and aim to place one copy in each of the zone in the grid (Figure 9). A four bit vector is added to the header of a packet to capture this information. Each bit corresponds to the presence of a packet in a zone. If (say) the bit for zone 1 is turned on, that means that one of the routers in the zone 1 has a copy of the data in it's local store. In our router, when a read reply packet is cached in the data store , the packet header is modified to turn on the current zone bit in the RLC stage. In the Figure 9, when a read reply packet enters the highlighted node in zone 4, it caries the information that it has been cached in zone 1 and zone 2 but not in zone 3 and zone 4. Since, the node seeing this packet is in zone 4, it decides it can cache this data and tries to cache it. Note that this logic could be implemented with a single bit but the scheme here illustrates the point better. In order for a router not to cache the data from it's own node we propose an optimization. Packets injected from the current node are not candidates for being added into the router's data store. The replacement scheme is as described in the independent caching. The zonal caching can lead to increase in contention for the cache space along the edges of the router as compared to an interior router. This scenario is not an issue in our experimental setup as all the routers except the corner routers are along the edge of a zone.

Table 1. Simulation setup

System simulation setup		
Number of cores		16
L1 I-Cache per core	capacity	16KB
	assoc	1 way
	access cycle	1 cycle
L1 D-Cache per core	capacity	16KB
	assoc	2 way
	access cycle	1 cycle
Unified L2 per core	capacity	256KB
	assoc	8 way
	access cycle	6 cycles

5 Experimental Setup

We use Simics [13], a full system simulator to model a 16-core CMP architecture. The system is modeled as an in-order core for the SPARC ISA running the Solaris 9 operating system. The cache configuration for the simulated system is shown in Table 1.

The SpecOMP [1] benchmark suite was run on this system and traces collected from their runs for addresses accessed. Each benchmark was marked with an initialization phase until it enters the main program. Caches were warmed up for 500 million instructions (executed per processor) at this stage. Traces were collected for the next 1 billion instructions (per processor). Our network on chip simulator modeled a 4 X 4 mesh with a 5 stage pipeline for the router. The simulator modeled the 'In-network cache coherence protocol' and the in network caching proposed in this work. The traces obtained for each benchmark were fed into the network simulator and the average latencies were observed. For all our results, the base case is a system with 'In Network Cache coherence'.

6 Experimental Results

The In-Network caching protocol proposed in this work is beneficial to the read latencies of accessing the memory. The write latencies are not directly modified by the protocol changes. However, the change in the traffic in the system changes them. We usually saw a very minor improvements for all of our experiments. The number we report in this work is the overall memory access latency for both reads and writes to the caches. Figure 10 shows the percentage improvement of memory access latency for the applications in the ideal caching scenario. The maximum performance improvement is shown by *apsi* (38%) and the least is shown by *art* (17%). On average the benefit shown by the benchmarks is 27%. It can be observed that there is a clear relation between the benefits shown by each benchmark and the number of accesses to shared lines. For example, *apsi* has the largest ratio of accesses to shared lines to accesses to private lines. Therefore, it benefits the most from the caching of shared lines. At the other end of the spectrum, the benchmarks *art* and *wupwise* have the two smallest values for the ratio

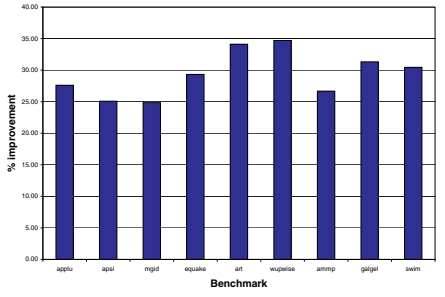


Fig. 10. Latency improvement using Ideal caching

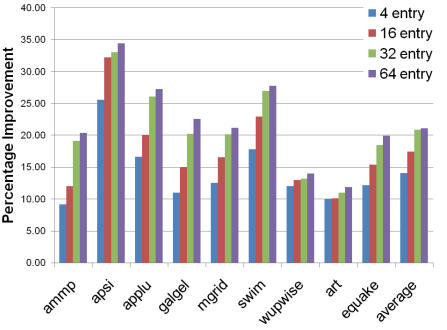


Fig. 11. Latency improvement using independent caching

of accesses to shared lines to accesses to private lines. Correspondingly, they also show the two lowest improvements when Ideal caching is employed.

Figure 11 shows the percentage improvement for each benchmark for four different local store sizes when Independent caching is employed over the base case when no In network caching is employed. The trend that can be observed is similar to that of the Ideal caching case. The benchmark *apsi* shows the most improvement in general and the most improvement when the number of entries is increased. The benchmark *wupwise* and *art* show the least improvement in general and the least relative improvement when the local store size is increased. The reasons for this are the same as the Ideal caching case. A more interesting comparison can be made when the improvements of the benchmarks *mgrid* and *swim* are studied. Both benchmarks access shared data more than 90% of the time. However, *swim* shows a disproportionately large performance improvement compared to *mgrid*. The reason behind this is that the percentage of shared data in *swim* is very small compared to *mgrid*. Therefore, an increase in the size of the local store allows *swim* to cache a greater portion of its *shared working set* compared to *mgrid*. Therefore, *swim* has a larger improvement compared to *mgrid*.

Figure 12 shows the benefits of Zonal caching and Independent caching for a local store of 4 entries. Clearly Zonal caching is more beneficial for all benchmarks compared to Independent caching. The average improvement of the zonal caching over the independent caching is 17%. The largest benefits over Independent Caching are shown by *apsi* and the lowest by *wupwise* and *art* for the reasons elaborated earlier. From the results shown in this figure, it is clear that Zonal caching should be the preferred approach to caching shared data.

To contrast the zonal and the independent schemes further, Figure 13 presents the percentage improvement of Zonal caching with a 4-entry local store normalized to the percentage improvement obtained by Independent caching with a 16-entry local store. We see that on average, Zonal caching with a 4-entry local store provides 97% of the performance improvements of Independent caching with 16-entry local store. Therefore, we can conclude that using Zonal caching with a 4-entry local store is as good as using Independent caching with a 16-entry local which in turn means that Zonal caching effectively provides a four-fold increase in local store capacity compared to Independent caching.

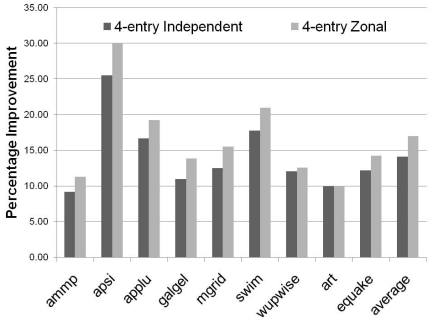


Fig. 12. Latency improvement using Zonal and Independent caching with a 4-entry data store

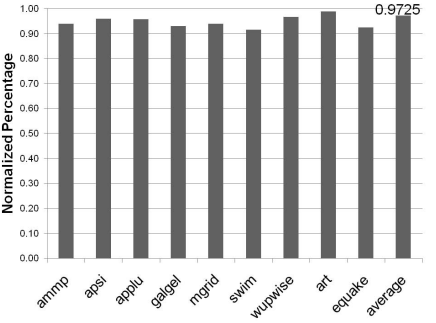


Fig. 13. Latency improvement offered by Zonal caching using a 4-entry local store normalized to the performance of a 16-entry data store using the independent scheme

7 Related Work

Power and performance consideration have motivated the use to chip multiprocessors [16]. As buses are not scalable, packet switched networks were proposed for on chip communication [5]. The basic router design is a five stage pipeline design is explained in [17]. This was further improved in several works including [15,2]. Recent research in the field of on-chip networks is surveyed in [4,7]. Kim et al. [9] aim to reduce the number of hops for packets in the network by using high radix networks. Kumar et al. [11] reduce the number of hops by skipping some routers along the way. Our work takes an alternate approach by bringing the required data closer to the requestor.

Liu et al. [12] also identify that the amount of shared data is low and the access to them frequent. As the number of concurrent accesses to a data block by the same processor is low, it is important that the data be placed where it is close to the processors that access it. They implement a fast *center cell* cache that resides in the center of the processor array. However, their work requires that the center cell be placed between all the processors in a 4-core CMP and is thus not scalable. In contrast, this work considers a distributed cache placed in the routers of the networks and the experiments are provided for a tiled 16-core system and is scalable to networks beyond this size.

Eisley et al. [6] propose (see section 2.1) to embed cache coherence information into the routers in the NoC itself. This allows data requests to be routed to the processor whose cache contains the data. Our work builds upon [6] by adding a small data store the routers that stores data being transported. These enhanced routers will be able to service requests for data that they hold which reduces the read latency times. Other works that optimize the network for the cache coherence protocol include [3,18].

Mizrahi et al. [14] implemented a cache coherence in network and also migrated data in the network. Their network is a multi stage interconnection network implemented as a tree. Our work focuses on a direct mesh of routers. None of the detailed pipeline proposals are presented in their work. Their evaluation scheme assumes a fixed number of writable variables and no particular benchmark unlike our work. Iyer et al. [8]. suggest a

similar technique of putting the caches in network switches. However, their implementation is for multiprocessor systems connected by generic network and not a solution for network-on-chip architecture. Since adding cache to a network switch doesn't effect the overall frequency of a system, their works lacks the implementation detail presented in our work.

8 Conclusion

This paper presented a novel approach to improving the effective performance of an NoC. Based on the premise that the amount of shared data in parallel applications is small and that the proportion of accesses to them are large, this work proposed to store shared data in the network fabric. The proposed approach involved adding a data store to the routers of the NoC. The paper presented a coherence protocol and a router microarchitecture to support the proposed scheme. Three caching policies were examined and the results show a significant decrease in the netowrk latency of the benchmark applications.

References

1. Aslot, V., Domeika, M.J., Eigenmann, R., Gaertner, G., Jones, W.B., Parady, B.: Specomp: A new benchmark suite for measuring parallel computer performance. In: Eigenmann, R., Voss, M.J. (eds.) WOMPAT 2001. LNCS, vol. 2104, pp. 1–10. Springer, Heidelberg (2001)
2. Bertozzi, D., Benini, L.: Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip. IEEE Circuits and Systems Magazine 4 (2004)
3. Bilir, E.E., Dickson, R.M., Hu, Y., Plakal, M., Sorin, D.J., Hill, M.D., Wood, D.A.: Multicast snooping: a new coherence method using a multicast address network. SIGARCH Comput. Archit. News 27(2), 294–304 (1999)
4. Bjerregaard, T., Mahadevan, S.: A survey of research and practices of network-on-chip. ACM Comput. Surv. 38(1), 1 (2006)
5. Dally, W.J., Towles, B.: Route Packets, Not Wires: On-Chip Interconnection Networks. In: Proc. of the 38th Design Automation Conference (DAC) (June 2001), `http://citeseer.ist.psu.edu/dally01route.html`
6. Eisley, N., Peh, L.-S., Li-Shang: In-network cache coherence. In: Proc. of the 39th Annual Intl. Symp. on Microarchitecture (MICRO) (2006)
7. Ivanov, A., Micheli, G.D.: The Network-on-Chip Paradigm in Practice and Research. Design & Test of Computers 22(5), 399–403 (2005)
8. Iyer, R.R., Bhuyan, L.N.: Design and evaluation of a switch cache architecture for cc-numa multiprocessors. IEEE Trans. Comput. 49(8), 779–797 (2000)
9. Kim, J., Balfour, J., Dally, W.: Flattened butterfly topology for on-chip networks. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2007, 1-5 December 2007, pp. 172–182 (2007)
10. Kim, J., Park, D., Nicopoulos, C., Narayanan, V., Das, C.: Performance enhancement through early release and buffer optimization in network-on-chip router architectures. In: Future Interconnects and Networks on Chip' at Design Automation and Test in Europe (DATE 2006) (2006)
11. Kumar, A., Peh, L.-S., Kundu, P., Jha, N.K.: Express virtual channels: towards the ideal interconnection fabric. In: ISCA 2007: Proceedings of the 34th annual international symposium on Computer architecture, pp. 150–161. ACM, New York (2007)

12. Liu, C., Sivasubramaniam, A., Kandemir, M., Irwin, M.: Enhancing l2 organization for cmps with a center cell. In: 20th International on Parallel and Distributed Processing Symposium, IPDPS 2006, 25-29 April 2006, p. 10 (2006)
13. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Gustav, H., Johan, H., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer 35(2), 50–58 (2002)
14. Mizrahi, H.E., Baer, J.L., Lazowska, E.D., Zahorjan, J.: Introducing memory into the switch elements of multiprocessor interconnection networks. SIGARCH Comput. Archit. News 17(3), 158–166 (1989)
15. Mullins, R.D., West, A.F., Moore, S.W.: Low-latency virtual-channel routers for on-chip networks. In: Proc. of the 31st Annual Intl. Symp. on Computer Architecture (ISCA), pp. 188–197 (2004), http://portal.acm.org/citation.cfm?id=998680.1006717
16. Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K., Chang, K.: The case for a single-chip multiprocessor. SIGOPS Oper. Syst. Rev. 30(5), 2–11 (1996)
17. Peh, L.-S., Dally, W.J.: A delay model and speculative architecture for pipelined routers. In: HPCA 2001: Proceedings of the 7th International Symposium on High-Performance Computer Architecture, Washington, DC, USA, p. 255. IEEE Computer Society Press, Los Alamitos (2001)
18. Stets, R., Dwarkadas, S., Kontothanassis, L., Rencuzogullari, U., Scott, M.: The effect of network total order, broadcast, and remote-write capability on network-based shared memory computing. In: Sixth International Symposium on High-Performance Computer Architecture, 2000. HPCA-6. Proceedings, pp. 265–276 (2000)

Parallel LDPC Decoding on the Cell/B.E. Processor

Gabriel Falcão[1], Leonel Sousa[2], Vitor Silva[1], and José Marinho[1]

[1] Instituto de Telecomunicações, University of Coimbra,
Polo II - Universidade de Coimbra
3030–290 Coimbra, Portugal
{gff,vitor,jmarinho}@co.it.pt
[2] INESC-ID/IST, Technical University of Lisbon
R. Alves Redol, n.9,
1000–129 Lisboa, Portugal
las@inesc-id.pt

Abstract. Low-Density Parity-Check (LDPC) codes are among the best error correcting codes known and have been recently adopted by data transmission standards, such as the second generation for Satellite Digital Video Broadcasting (DVB-S2) and WiMAX. LDPC codes are based on sparse parity-check matrices and use message-passing algorithms, also known as belief propagation, which demands very intensive computation. For that reason, VLSI dedicated architectures have been proposed in the past few years, to achieve real-time processing. This paper proposes a new flexible and programmable approach for LDPC decoding on a heterogeneous multicore Cell Broadband Engine (Cell/B.E.) architecture. Very compact data structures were developed to represent the bipartite graph for both regular and irregular LDPC codes. They are used to map the irregular behavior of the Sum-Product Algorithm (SPA) used in LDPC decoding into a computing model that expresses parallelism and locality of data by decoupling computation and memory accesses. This model can be used in general for exploiting capabilities of modern multicore architectures. For the Cell/B.E., in particular, stream-based programs were developed for simultaneous multicodeword LDPC decoding by using SIMD features and a low-latency DMA-based data communication mechanism between processors. Experimental results show significant throughputs that compare well with state-of-the-art VLSI-based solutions.

1 Introduction

Low-Density Parity-Check (LDPC) codes are powerful error correcting codes (ECC) proposed by Robert Gallager in 1962 [1] which allow to work very close to the Shannon limit. Rediscovered by Mackay and Neal in 1996 [2], they have inspired the scientific community to develop efficient LDPC coding solutions [3], [4], [5], [6], [7] for data communication systems. They have recently been adopted by the DVB-S2 standard, WiMAX (802.16e), Wifi (802.11n), 10Gbit Ethernet

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 389–403, 2009.

(802.3an), and other new emerging standards for communication and storage applications. LDPC codes are linear (N, K) block codes [8] defined by sparse binary parity check **H** matrices of dimension $(N - K) \times N$. They are usually represented by bipartite graphs formed by Bit Nodes (BNs) and Check Nodes (CNs), and linked by bidirectional edges, also called *Tanner* graphs [9]. LDPC decoders, and in particular the Sum-Product Algorithm (SPA), require very intensive floating-point computation. For this reason, dedicated VLSI hardware solutions based on fixed-point arithmetic in the logarithmic domain [10] have been proposed over the last few years. Nevertheless, a software-based, flexible, scalable and low-cost solution to this computationally intensive problem would be highly desirable.

In line with this perspective, manycore architectures have recently been proposed for implementing LDPC decoders under a multithreaded approach based on Graphics Processing Units (GPUs) [11], where multiple sub-partitions of the *Tanner* graph are distributed into different threads on the GPU grid to be processed on different processors, using a high number of threads as a technique to hide latency. These implementations have been performed using either a dedicated GPU programming interface such as the recent Computer Unified Device Architecture (CUDA) from NVIDIA [12], or the generic Caravela programming interface tool [13].

Pushed by the games industry, the STI Cell Broadband Engine (Cell/B.E.) architecture [14] is a joint venture involving Sony, Toshiba and IBM, characterized by a heterogeneous multicore architecture. It is composed by one main 64-bit PowerPC Processor Element (PPE) that communicates with eight Synergistic Processor Elements (SPEs) each having 256 KByte of local memory for data and source code, and a vectorized 128-bit wide SIMD oriented architecture. Data transfers between PPE and SPE are performed efficiently by using Direct Memory Accesses (DMA).

This work proposes a novel approach to the SPA algorithm oriented for LDPC decoding [15] on the Cell multicore architecture [16], supported on a new compact data structure that suits data-parallelism and efficiently represents the *Tanner*

(8 bit nodes checked by 4 check node equations)

$$H = \begin{array}{c} \begin{array}{cccccccc} B_0 & B_1 & B_2 & B_3 & B_4 & B_5 & B_6 & B_7 \end{array} \\ \left[\begin{array}{cccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right] \end{array} \begin{array}{c} C_0 \\ C_1 \\ C_2 \\ C_3 \end{array}$$

Fig. 1. An **H** matrix example with dimensions 4×8

graph. We propose to run the LDPC decoder on this heterogeneous architecture by putting the main PPE in charge of the overall control, while offloading the intensive computation of a considerable number of codewords across the several SPEs [17]. Local memory on each SPE is exploited, while at the same time data communication overhead is minimized. The Cell/B.E. architecture is also able to perform floating-point arithmetic operations, which may improve accuracy and produce lower Bit Error Rates (BER), namely regarding to VLSI fixed-point LDPC decoders [10]. The compact data representation and the new algorithm based on a specific type of stream-based computing model, that exploits the use of SIMD instructions [18] on the Cell/B.E., are the main contributions of this paper.

This paper is organized as follows. The next section addresses the properties of the SPA algorithm used in LDPC decoding. Section 3 describes the parallel approach used in this work. Section 4 presents the architecture details and the SPA flooding schedule algorithm specifically developed for SIMD computing on the Cell processor. Section 5 reports experimental results obtained for LDPC decoding on the Cell/B.E. architecture. The last section concludes the paper.

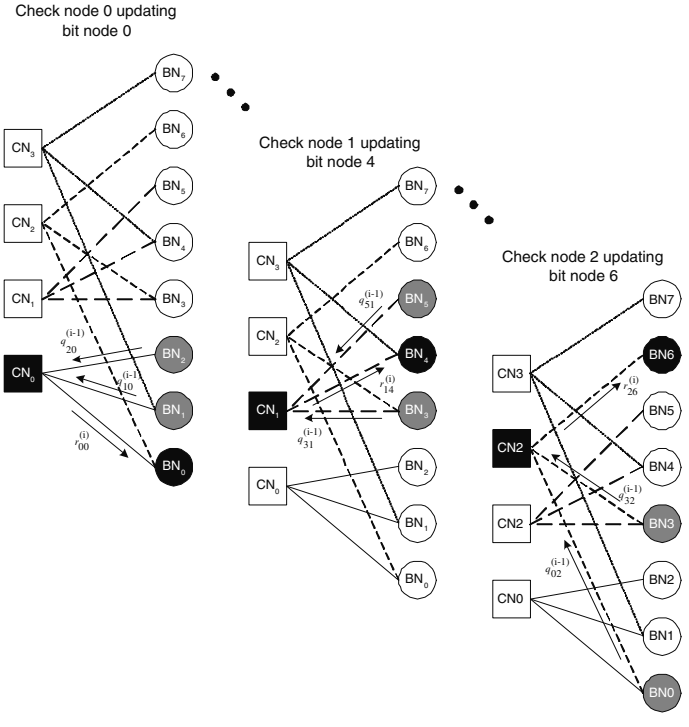


Fig. 2. Tanner graph representation of the **H** matrix in figure 1. The example shows messages being exchanged between CN_m and BN_n. A similar representation applies for Check Nodes message updating.

2 SPA for LDPC Decoding

LDPCs are linear (N, K) block codes defined by sparse binary parity check $\mathbf{H}$ matrices of dimension $(N - K) \times N$. They are usually represented by bipartite graphs, also called *Tanner* graphs [9], formed by Bit Nodes (BNs) and Check Nodes (CNs) linked by bidirectional edges. The SPA is a very efficient algorithm [8] used for LDPC decoding. It is based on the belief propagation between nodes connected as indicated by the *Tanner* graph edges (see an example in figure 2). As proposed by Gallager, the Sum-Product Algorithm (SPA) operates on probabilities [1] [2] [8].

Given a (N, K) LDPC code, we assume BPSK modulation which maps a codeword $\mathbf{c} = (c_1, c_2, \cdots, c_n)$, into the sequence $\mathbf{x} = (x_1, x_2, \cdots, x_n)$, according to $x_i = (-1)^{c_i}$. Then, $\mathbf{x}$ is transmitted through an additive white Gaussian noise (AWGN) channel given rise to the received sequence $\mathbf{y} = (y_1, y_2, \cdots, y_n)$, with $y_i = x_i + n_i$, where n_i is a random variable with zero mean and variance $\sigma^2 = N_0/2$.

SPA is depicted in Algorithm 1 [8]. It is mainly described by two horizontal and vertical intensive processing blocks, respectively defined by equations (1), (2) and (3), (4). The first one calculates the message update from each CN_m to BN_n, considering accesses to $\mathbf{H}$ on a row basis. It indicates the probability of BN_n being 0 or 1. Figure 2 exemplifies, for a particular 4×8 $\mathbf{H}$ matrix, BN_0 being updated by CN_0, BN_4 updated by CN_1 and BN_6 by CN_2. Similarly, the second major block computes messages sent from BN_n to CN_m, assuming accesses on a column basis. The iterative procedure is stopped if the decoded word $\hat{\mathbf{c}}$ verifies all parity check equations of the code $\hat{\mathbf{c}}\,\mathbf{H}^T = \mathbf{0}$, or a maximum number of iterations is reached, in which case no codeword is detected.

The irregular memory access patterns in LDPC decoding represent a challenge to the efficiency of the proposed solution and are depicted in figure 3. The access to different nodes in the *Tanner* graph is defined by the $\mathbf{H}$ matrix and should favor randomization in order to allow good coding gains. For that reason, the data structures developed and represented in figure 6 try to minimize that effect, by grouping data computed in the same step. A global irregular access pattern is translated into several partial regular access patterns.

3 Parallelization Approach

The parallelization approach proposed for the LDPC decoder is explained in the context of the Cell/B.E. architecture. The LDPC decoder processes on an iterative basis. The PPE reads information y_n from the input channel and produces the probabilities p_n as indicated in Algorithm 1. After receiving the corresponding p_n values, each SPE performs two steps: *(i)* computes kernel 1 and kernel 2 alternately using SIMD instructions; *(ii)* sends the final results back to the PPE which replaces new data to be sent to the SPE. Data is communicated between the PPE and the SPEs by efficient Direct Memory Accesses (DMA).

The parallel LDPC decoder explores data-parallelism by applying the same algorithm to 4 codewords on each SPE (see figure 5). Data are represented

Algorithm 1. SPA

1: $q_{mn}^{(0)}(0) = 1 - p_n \;\; ; q_{mn}^{(0)}(1) = p_n;$

$p_n = \frac{1}{1+e^{\frac{2y_n}{\sigma^2}}} \;\; ; \frac{E_b}{N_0} = \frac{N}{2K\sigma^2}$ {Initialization}

2: **while** ($\hat{\mathbf{c}}\,\mathbf{H}^T \neq \mathbf{0} \wedge \mathrm{i} < \mathrm{I}$) {c-decod. word;I-Max no. iterations.} **do**

3: {For each node pair (BN_n, CN_m), corresponding to $\mathbf{H_{mn}} = \mathbf{1}$ in the parity check matrix $\mathbf{H}$ of the code **do**:}

4: {Compute the message sent from CN_m to BN_n, that indicates the probability of BN_n being 0 or 1:}

(Kernel 1)

$$r_{mn}^{(i)}(0) = \frac{1}{2} + \frac{1}{2} \prod_{n' \in N(m) \backslash n} \left(1 - 2q_{n'm}^{(i-1)}(1)\right) \quad (1)$$

$$r_{mn}^{(i)}(1) = 1 - r_{mn}^{(i)}(0) \quad (2)$$

{where $N(m)\backslash n$ represents BN's connected to CN_m excluding BN_n.}

5: {Compute message from BN_n to CN_m, which indicates the probability of BN_n being 0 or 1:}

(Kernel 2)

$$q_{nm}^{(i)}(0) = k_{nm}(1 - p_n) \prod_{m' \in M(n) \backslash m} r_{m'n}^{(i)}(0) \quad (3)$$

$$q_{nm}^{(i)}(1) = k_{nm} p_n \prod_{m' \in M(n) \backslash m} r_{m'n}^{(i)}(1) \quad (4)$$

{where k_{nm} are chosen to ensure $q_{nm}^{(i)}(0) + q_{nm}^{(i)}(1) = 1$, and $M(n)\backslash m$ is the set of CN's connected to BN_n excluding CN_m.}

6: {Compute the *a posteriori* pseudo-probabilities:}

$$Q_n^{(i)}(0) = k_n(1 - p_n) \prod_{m \in M(n)} r_{mn}^{(i)}(0)$$

$$Q_n^{(i)}(1) = k_n p_n \prod_{m \in M(n)} r_{mn}^{(i)}(1)$$

{where k_n are chosen to guarantee $Q_n^{(i)}(0) + Q_n^{(i)}(1) = 1$.}

7: {Perform hard decoding} $\forall n$,

$$\hat{c}_n^{(i)} = \begin{cases} 1 & \Leftarrow \quad Q_n^{(i)}(1) > 0.5 \\ 0 & \Leftarrow \quad Q_n^{(i)}(1) < 0.5 \end{cases}$$

8: **end while**

as 32-bit precision floating-point numbers. The proposed LDPC decoder suits scalability and for that reason it can be easily adopted by future manycore generations of the architecture with a higher number of SPEs, namely the next Cell generation, which is expected to have more SPEs available for intensive

processing. In that case, it will be able to decode more codewords simultaneously, increasing the efficiency and throughput of the decoder. The PPE controls the main tasks, offloading the intensive processing to the SPEs, where the processing is then distributed over several threads. Data is loaded from the main memory into the local storage (LS) memory on the SPEs via DMA units. The data structures that define the *Tanner* graph and the SPA are also loaded into the LS on the SPEs where the processing is performed, and at the end the processed data is returned to the PPE, which concludes the processing of the current codewords and starts the new ones. Each SPE runs independently of the other SPEs.

A complete iteration inside the SPE is processed in two phases: *(i)* kernel 1 computes data according to equations (1) and (2) defined in Algorithm 1, where the processing is performed in a row-major order. The data structure designed to represent r_{mn} in order to perform this task is depicted in figure 6 a). It can be seen in this figure that data related to BNs common to a CN equation is stored in contiguous memory positions to optimize processing; and *(ii)* kernel 2 processes data on a column-major order, defined by equations (3) and (4) in Algorithm 1. The q_{nm} data structure used in this case is depicted in figure 6 b). The SPE accesses data in a row or column-major order, depending on the kernel that is being processed at the time.

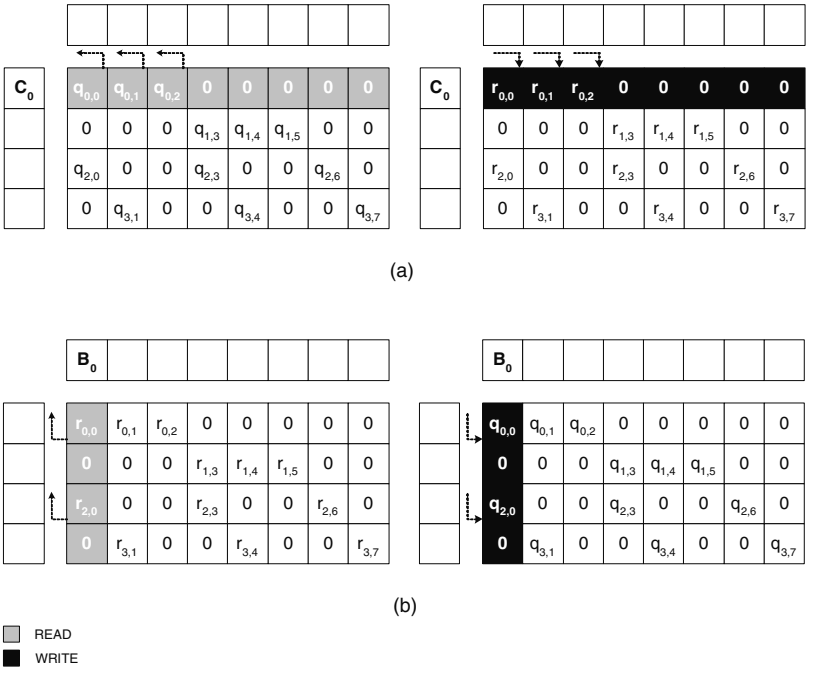


Fig. 3. Illustration of irregular memory accesses for the example in figure 2

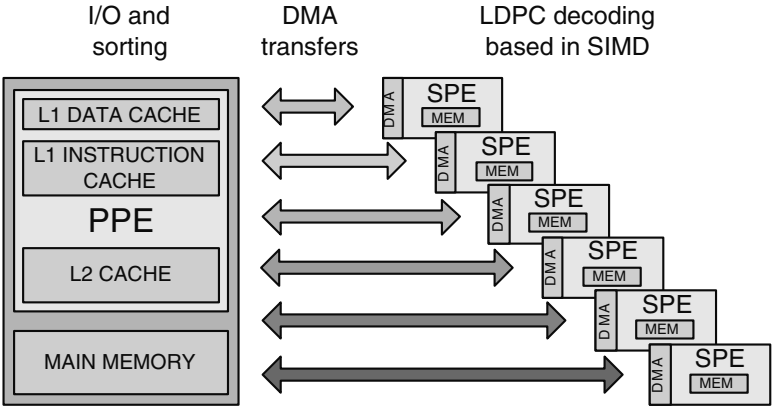


Fig. 4. Parallelization model for an LDPC decoder on the Cell/B.E. architecture

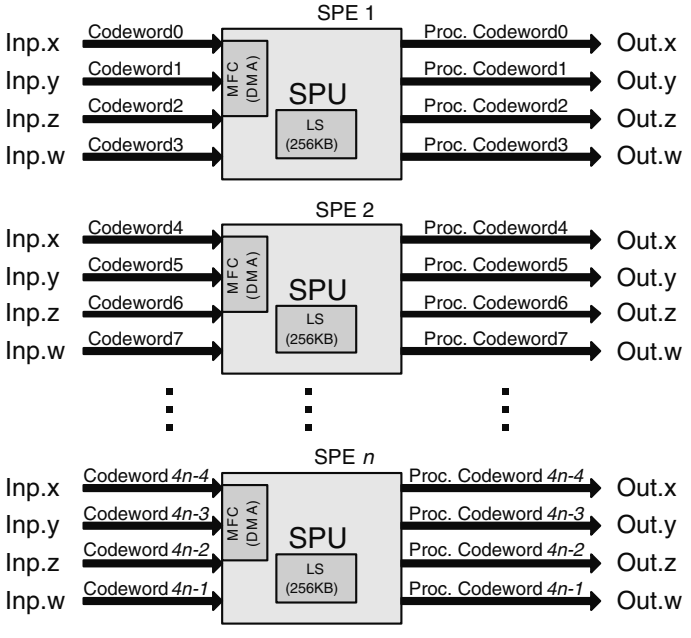


Fig. 5. Detail of the parallelization model developed for an LDPC simultaneously decoding several codewords on the SPEs

Kernel 1 performs the horizontal processing according to the *Tanner* graph, and the $r_{mn}^{(i)}$ data defined in section 2 is updated for iteration i. The data is initially transferred to the LS of the SPE by performing a DMA transaction, and its access organization maximizes data reuse, because a CN updating BNs reads common information from several BNs that share data among them. Figure 6 b) shows the data structures that hold the q_{nm} values to be read and also the

corresponding indexes of the r_{mn} elements in figure 6 a) that they are going to update. As depicted in figure 6, BNs and CNs associated with the same r_{mn} or q_{nm} equation are represented in contiguous blocks of memory.

In kernel 2 data is processed in a column-major order, which is defined by vertical processing. According to the *Tanner* graph, each BN updates all the CNs connected to it and holds the addresses necessary to complete the update of all $q_{nm}^{(i)}$ data for iteration i. Once again maximum data reusing is achieved, but this time among data belonging to the same column of the **H** matrix, as depicted in figures 2 and 6 b).

The computation is performed in the SPE for a predefined number of iterations. One of the purposes of this work is to evaluate the performance, namely the throughput, over different computing models. Pursuing this goal, we decided to develop a solution where the number of iterations is fixed to allow a fair comparison between different approaches, where the processing workload is known *a priori* and the same for all environments. This is why, at the end of an iteration, we don't check if the decoder produces a valid codeword, which would cause the decoding processing to stop. Nevertheless, this operation represents a negligible overhead. When all the BNs and CNs are updated after the final iteration, the SPE activates a DMA transaction and sends data back to the main memory, signalizing the PPE to conclude the processing. As the DMA finishes transferring the data, synchronization points are introduced to allow data buffers reuse.

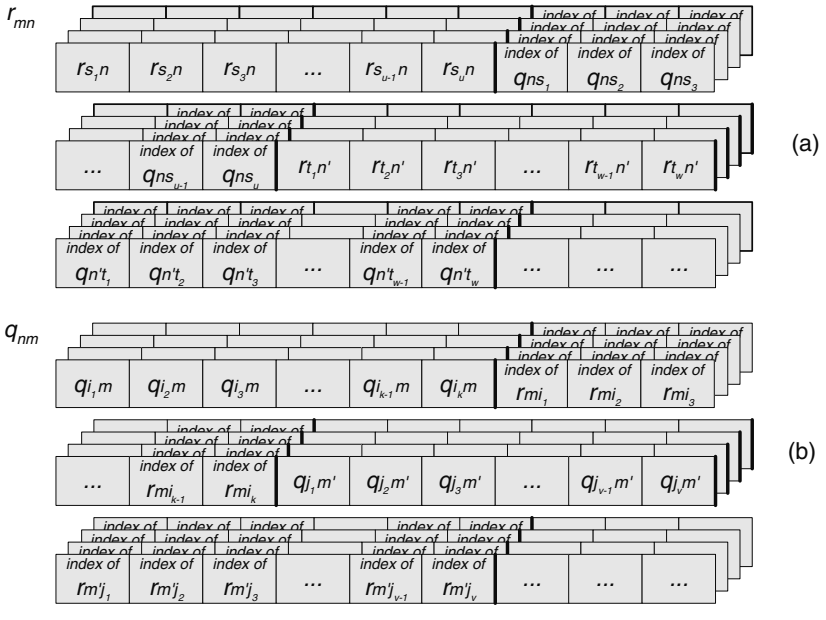


Fig. 6. Segment of the SIMD vectorized data structures in memory to represent: a) r_{mn} messages associated to BNn and BNn' nodes; and b) q_{nm} messages associated to CNm and CNm' nodes

4 SPA LDPC Decoding on the Cell/B.E. Architecture

The parallel approach proposed in the previous section is applied to the LDPC decoder on the Cell/B.E. processor. The recursive updating mechanism applied to BNs and CNs is performed in a sequence of pairs $\{BN, CN\}$ and tested for a number of iterations ranging from 10 to 100, using single precision floating-point arithmetic. Synchronization between the PPE and the SPEs is performed using mailboxes.

This approach tries to explore data-parallelism and data locality while performing the partitioning and mapping of the algorithm and data structures over the multiple cores, and at the same time minimizes delays caused by latency and synchronization.

4.1 PPE

The part of the algorithm that executes on the PPE side is presented in Algorithm 2. We force the PPE to communicate with only one SPE, which is called the MASTER SPE, and performs the control over the remaining SPEs. This is more efficient than putting the PPE controlling all the SPEs.

The PPE receives the y_n information from the channel and calculates probabilities p_n, after which it sends a *NEW_WORD* message to the MASTER SPE. Then, it waits for the download of all p_n probabilities to the SPEs and for the processing to be completed in each one of them.

Finally, when all the iterations are completed, the MASTER SPE sends an *END_DECODE* message to the PPE to conclude the current decoding process and get ready to start processing a new word.

Algorithm 2. PPE side of the algorithm

```
1: for th_ctr = 1 to NSPEs: do
2:     Create th_ctr thread
3: end for
4: repeat
5:     Receives y_n from the channel and calculates p_n probabilities
6:     Send msg NEW_WORD to MASTER SPE
Ensure: Wait until mail is received (SPE[i].mailboxcount > 0) from MASTER SPE
7:     msg = SPE[i].mailbox (received msg END_DECODE from MASTER SPE)
8: until true
```

4.2 SPE

The SPEs are used for the intensive processing task of updating all BNs and CNs by executing kernel 1 and kernel 2 (in Algorithm 1), in each decoding iteration. Each thread running on the SPEs accesses the main memory by using DMA and computes data according to the *Tanner* graph, as defined in the **H** matrix (see the example in figure 2). The MASTER SPE side of the procedure is described in

Algorithm 3. MASTER SPE side of the algorithm

```
 1: repeat
Ensure: Read mailbox (waiting a NEW_WORD mail from PPE)
 2:   Broadcast msg NEW_WORD to all other SPEs
 3:   Get p_n probabilities
 4:   for i = 1 to N_Iter: do
 5:     Compute r_mn
 6:     Compute q_nm
 7:   end for
 8:   Put final Q_n values on the PPE
Ensure: Read mailbox (waiting an END_DECODE mail from all other SPEs)
 9:   Send msg END_DECODE to PPE
10: until true
```

Algorithm 3. The *Get* operation is adopted to represent a communication PPE → SPE, while the *Put* operation is used for communication in the opposite direction.

We initialize the process and start an infinite loop, waiting for communications to arrive from the PPE (in the case of the MASTER SPE), or from the MASTER SPE (for all other SPEs). In the MASTER SPE, the only kind of message expected from the PPE is a *NEW_WORD* message. When a *NEW_WORD* message is received, the MASTER SPE broadcasts a *NEW_WORD* message to all other SPEs and loads p_n probabilities associated to itself. After receiving these messages, each one of the other SPEs also gets its own p_n values.

The processing terminates when the number of iterations is reached and an *END_DECODE* mail is sent by all SPEs to the MASTER SPE, which immediately notifies the PPE with an *END_DECODE* message.

Algorithm 4. SLAVE SPE side of the algorithm

```
 1: repeat
Ensure: Read mailbox (waiting a NEW_WORD mail from MASTER SPE)
 2:   Get p_n probabilities
 3:   for i = 1 to N_Iter: do
 4:     Compute r_mn
 5:     Compute q_nm
 6:   end for
 7:   Put final Q_n values on the PPE
 8:   Send msg END_DECODE to MASTER SPE
 9: until true
```

The intensive part of the computation in LDPC decoding on the Cell/B.E. architecture takes advantage of the processing power and SIMD instruction set available on the SPEs, which means that several codewords are decoded simultaneously.

5 Experimental Results

To evaluate the performance of the proposed LDPC decoder, the Cell/B.E. was programmed using: (*i*) the PPE alone which is denoted by serial mode in figure 7; and (*ii*) the complete set of PPE + six SPE processors denoted by parallel mode in the same figure. The Cell/B.E. is included in a PlayStation 3 (PS3) platform, which restricts the number of available SPEs to 6, from a total of 8.

Table 1. Experimental setup

	Serial mode	**Parallel mode**	
Platform	PPE	STI Cell/B.E.	
Language	C	C	
OS	Linux (Fedora) kernel 2.6.16		
		PPE	SPE
Clock frequency	3.2GHz	3.2GHz	3.2GHz
Memory	256MB	256MB	256KB

The serial mode uses a dual thread approach and exploits SIMD instructions. It should be noted that by performing the comparison based on the time per bit decoded, the serial solution that uses only the PPE is slower than the execution on a single SPE, because the PPE accesses the slow main memory, while the SPE accesses the fast LS. On the parallel approach the experimental results were also obtained using SIMD instructions, which are responsible for the intensive decoding part of the algorithm, where 4 floating-point elements are packed and operated in a single instruction, making it possible to decode 4 codewords in parallel on each SPE. All the processing times were measured for decoders performing a number of iterations ranging from 10 to 100.

The experimental setup for the two modes performing LDPC decoding is depicted in table 1. The PPE runs at 3.2GHz and has 256MByte of RAM memory, and each of the 6 SPEs runs at 3.2GHz and has 256 KByte of local memory.

All matrices under test were run on both models and are presented in table 2. Matrices A, B and C are regular with $rate = 1/2$ and present small to medium workloads. Table 2 shows the dimensions of the matrices under test and the corresponding sizes that their equivalent data structures (depicted in figure 6)

Table 2. Parity check matrices under test

Matrix	**Size**	**Edges**	**Edges/row**	**Data structures memory occupancy on the LS of a SPE**
A	128×256	768	6	35840 Bytes
B	252×504	1512	6	70560 Bytes
C	512×1024	3072	6	143360 Bytes

occupy inside the memory of a SPE. The maximum regular code allowed is limited by the LS size, according to:

$$N \times Edges/row \times 18 + 32 \times N + Program\_Size \leq 256KByte. \quad (5)$$

Figure 7 presents the decoding times for matrices A to C. They relate to the execution time needed for the Cell/B.E. to decode a codeword on the parallel mode and compare it against the processing time required to decode the same codeword on the serial mode. The decoding time for matrix C shows that on the parallel mode the Cell/B.E. takes approximately $14.7\mu s$ to decode a 1024 bits codeword in 10 iterations, against $244.7\mu s$ on the serial mode. Considering that the Cell/B.E. performs concurrent processing on the six SPEs, each using SIMD to process several codewords simultaneously, we conclude that it achieves a throughput of 69.5 Mbps. Comparing this throughput with that obtained on the serial mode, the speedup achieved for matrix C running on the Cell/B.E. is 16. Experimental evaluations show that the same algorithm applied only to one SPE (also using SIMD) has a global processing time less than 1% faster than the full version that uses six SPEs. In fact, due to the existence of low-latency DMA hardware and fast mailbox procedures, the impact of data transfers in this model is so low, that the overhead in communications introduced by using a higher number of SPEs, as expected in the next generations of the Cell/B.E. architecture, would be marginal.

5.1 Discussion

Memory access constraints represent a serious challenge in multicore architectures. The parallel approach proposed in this paper for LDPC decoding, based

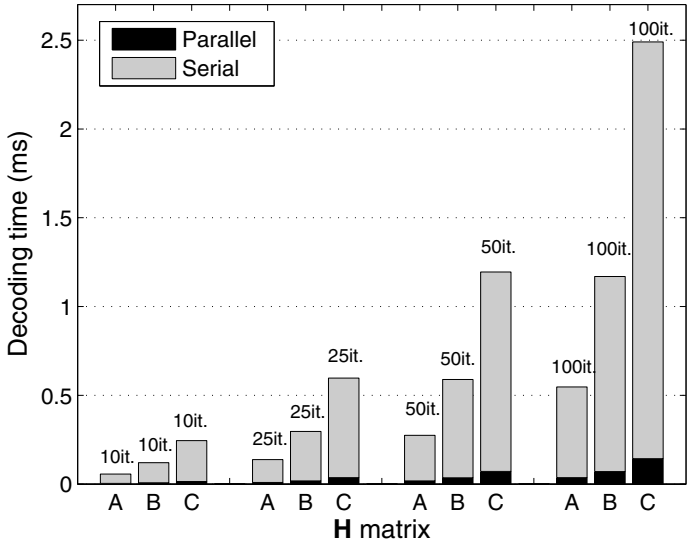


Fig. 7. LDPC decoding times on the Cell/B.E. in serial and parallel modes

in intensive irregular accesses to memory, tries to minimize that effect. In order to analyze the obtained throughputs, we propose the following mathematical model, which assumes a hypothetical upper bound throughput T_{ma}, as if the algorithm only performed memory accesses (and no arithmetic operations) on the local SPE:

$$T_{ma} = \frac{4 \times n \times P \times W_m}{6 \times Edges \times N_{iter}}, \tag{6}$$

where n denotes the codeword length, P the number of SPEs, N_{iter} is the number of iterations performed, *Edges* the number of *Tanner* graph edges of the code and W_m represents the SPE memory bandwidth ($W_m \simeq 3.01$ Gword/s[1] was experimentally obtained). Denoting α as the ratio between the global throughputs T (depicted in table 3) and T_{ma}, as $T = \alpha \times T_{ma}$, it was found that $\alpha < 1$ and approximately constant (equal to 0.17 with negligible variation) for all code lengths and rates. This mathematical model shows that the global throughputs in table 3 are mainly limited by the SPE processing time rather than bottlenecks in memory accesses, which is a very important aspect when using an architecture based in multicore processing such as the present one. The T_{ma} model fits the experimental results with adequate precision.

Table 3. Throughputs obtained in the parallel mode (Mbps)

Matrix	10 iter.	25 iter.	50 iter.	100 iter.
A	68.5	28.0	14.2	7.1
B	69.1	28.3	14.2	7.2
C	69.5	28.4	14.3	7.2

The results reported in table 3 for the SPA algorithm in the proposed Cell/B.E. architecture are better than those reported in [7], that implements LDPC decoding with an algorithm computationally less intensive (Min-Sum) than the SPA on a hardware platform with similar characteristics. At the same time, our results are only slightly lower than those reported in VLSI dedicated solutions [19], and also for the less complex Min-Sum algorithm.

6 Conclusion

This paper proposes a novel parallel approach for LDPC decoding on multicore architectures. Compact and vectorized data structures to represent the exchanged messages between connected nodes (BN_n, CN_m) on the *Tanner* graph of an LDPC decoder are presented. These data structures allow a significant reduction of both the memory space and the processing time necessary for LDPC decoding. The SPA was adapted for stream computing suiting LDPC decoding on the Cell/B.E. architecture (CBEA). Significant speedups superior to 15 and

[1] Giga 128-bit words per second.

throughputs near 70 Mbps are reported. The solution proposed in this paper is scalable to future generations of the Cell/B.E. architecture, that are expected to have more SPEs, being able to produce even better performances, with a minimal increase of overhead in data communications between processors. It provides a low-cost and flexible software-based multicore alternative with performances that compare well to the expensive, non-scalable and hardware-dedicated VLSI LDPC decoder solutions.

References

1. Gallager, R.G.: Low-Density Parity-Check Codes. IRE Transactions on Information Theory IT-8, 21–28 (1962)
2. Mackay, D.J.C., Neal, R.M.: Near Shannon Limit Performance of Low Density Parity Check Codes. IEE Electronics Letters 32(18), 1645–1646 (1996)
3. Dielissen, J., Hekstra, A., Berg, V.: Low cost LDPC decoder for DVB-S2. In: Proceedings of Design, Automation and Test in Europe (DATE 2006), pp. 1–6 (2006)
4. Zhang, T., Parhi, K.: Joint (3,k)-regular LDPC code and decoder/encoder design. IEEE Tansactions on Signal Processing 52(4), 1065–1079 (2004)
5. Gomes, M., Falcão, G., Silva, V., Ferreira, V., Sengo, A., Falcão, M.: Flexible Parallel Architecture for DVB-S2 LDPC Decoders. In: IEEE GLOBECOM 2007, Washington, DC, USA (2007)
6. Verdier, F., Declercq, D.: A low-cost parallel scalable FPGA architecture for regular and irregular LDPC decoding. IEEE Transactions on Communications 54(7), 1215–1223 (2006)
7. Seo, S., Mudge, T., Zhu, Y., Chakrabarti, C.: Design and Analysis of LDPC Decoders for Software Defined Radio. In: IEEE Workshop on Signal Processing Systems, pp. 210–215 (2007)
8. Lin, S., Costello, D.J.: Error Control Coding, 2nd edn. Prentice Hall, Englewood Cliffs (2004)
9. Tanner, R.: A Recursive Approach to Low Complexity Codes. IEEE Transactions on Information Theory 27, 533–547 (1981)
10. Ping, L., Leung, W.K.: Decoding Low Density Parity Check Codes with Finite Quantization Bits. IEEE Communications Letters 4(2), 62–64 (2000)
11. Falcão, G., Sousa, L., Silva, V.: Massive Parallel LDPC Decoding on GPU. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP 2008), Salt Lake City, Utah, USA, pp. 83–90 (2008)
12. NVIDIA, `http://developer.nvidia.com/object/cuda.html`
13. Yamagiwa, S., Sousa, L.: Caravela: A Novel Stream-based Distributed Computing Environment. IEEE Computer 40(5), 76–83 (2007)
14. International Business Machines Corporation, CELL Broadband Engine Architecture (2006)
15. Hu, X.-Y., Eleftheriou, E., Arnold, D.-M., Dholakia, A.: Efficient Implementations of the Sum-Product Algorithm for Decoding LDPC Codes. In: IEEE GLOBECOM 2001, vol. 2, pp. 1063–1067 (2001)
16. Hofstee, H.: Power Efficient Processor Architecture and the Cell Processor. In: 11th International Symposium on High-Performance Computer Architectures (HPCA), pp. 258–262 (2005)

17. Sony Computer Entertainment Incorporated, SPU C/C++ Language Extensions (2005)
18. International Business Machines Corporation, Synergistic Processor Unit Instruction Set Architecture (2007)
19. Liu, C.-H., Yen, S.-W., Chen, C.-L., Chang, H.-C., Lee, C.-Y., Hsu, Y.-S., Jou, S.-J.: An LDPC Decoder Chip Based on Self-Routing Network for IEEE 802.16e Applications. IEEE Journal of Solid-State Circuits 43(3), 684–694 (2008)

Parallel H.264 Decoding on an Embedded Multicore Processor

Arnaldo Azevedo[1], Cor Meenderinck[1], Ben Juurlink[1], Andrei Terechko[2], Jan Hoogerbrugge[2], Mauricio Alvarez[3], and Alex Ramirez[3,4]

[1] Delft University of Technology, Delft, The Netherlands
{Azevedo,Cor,Benj}@ce.et.tudelft.nl
[2] NXP, Eindhoven, The Netherlands
{andrei.terechko,jan.hoogerbrugge}@nxp.com
[3] Technical University of Catalonia (UPC), Barcelona, Spain
alvarez@ac.upc.edu
[4] Barcelona Supercomputing Center (BSC), Barcelona, Spain
alex.ramirez@bsc.es

Abstract. In previous work the 3D-Wave parallelization strategy was proposed to increase the parallel scalability of H.264 video decoding. This strategy is based on the observation that inter-frame dependencies have a limited spatial range. The previous results, however, investigate application scalability on an idealized multiprocessor. This work presents an implementation of the 3D-Wave strategy on a multicore architecture composed of NXP TriMedia TM3270 embedded processors. The results show that the parallel H.264 implementation scales very well, achieving a speedup of more than 54 on a 64-core processor. Potential drawbacks of the 3D-Wave strategy are that the memory requirements increase since there can be many frames in flight, and that the latencies of some frames might increase. To address these drawbacks, policies to reduce the number of frames in flight and the frame latency are also presented. The results show that our policies combat memory and latency issues with a negligible effect on the performance scalability.

1 Introduction

The demand for computational power increases continuously as the consumer market forecasts new applications such as Ultra High Definition (UHD) video [1], 3D TV [2], and real-time High Definition (HD) video encoding. In the past this demand was mainly satisfied by increasing the clock frequency and by exploiting more instruction-level parallelism (ILP). Due to the inability to increase the clock frequency much further because of thermal constraints and because it is difficult to exploit more ILP, multicore architectures have appeared on the market.

This new paradigm relies on the existence of sufficient thread-level parallelism (TLP) to exploit the large number of cores. Techniques to extract TLP from applications will be crucial to the success of multicores. This work investigates the exploitation of the TLP available in an H.264 video decoder on an embedded

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 404–418, 2009.

multicore processor. H.264 was chosen due to its high computational demands, wide utilization, and development maturity and the lack of "mature" future applications. although a 64-core processor is not required to decode a Full High Definition (FHD) video in real-time. Real-time encoding remains a problem and decoding is part of encoding. Furthermore, emerging applications such as 3DTV are likely to be based on current video coding methods [2].

In previous work [3] we have proposed the 3D-Wave parallelization strategy for H.264 video decoding. It has been shown that the 3D-Wave strategy potentially scales to a much larger number of cores than previous strategies. However, the results presented there are analytical, analyzing how many macroblocks (MBs) could be processed in parallel assuming infinite resources, no communication delay, infinite bandwidth, and a constant MB decoding time. In other words, our previous work is a limit study.

Compared to [3], we make the following contributions:

- We present an implementation of the 3D-Wave strategy on an embedded multicore consisting of up to 64 TM3270 processors. Implementing the 3D-Wave turned out to be quite challenging. It required to dynamically identify inter-frame MB dependencies and handle their thread synchronization, in addition to intra-frame dependencies and synchronization. This led to the development of a subscription mechanism where MBs subscribe themselves to a so-called *Kick-off List* (KoL) associated with the MBs they depend on. Only if these MBs have been processed, processing of the dependent MBs can be resumed.
- A potential drawback of the 3D-Wave strategy is that the latency may become unbounded because many frames will be decoded simultaneously. A policy is presented that gives priority to the oldest frame so that newer frames are only decoded when there are idle cores.
- Another potential drawback of the 3D-Wave strategy is that the memory requirements might increase because of large number of frames in flight. To overcome this drawback we present a frame scheduling policy to control the number of frames in flight.

Parallel implementations of H.264 decoding and encoding have been described in several papers. Rodriguez et al. [4] implemented an H.264 encoder using Group of Pictures (GOP)- (and slice-) level parallelism on a cluster of workstations using MPI. Although real-time operation can be achieved with such an approach, the latency is very high.

Chen et al. [5] presented a parallel implementation that decodes several B frames in parallel. However, even though uncommon, the H.264 standard allows to use B frames as reference frames, in which case they cannot be decoded in parallel. Moreover, usually there are no more than 2 or 3 B frames between P frames. This limits the scalability to a few threads. The 3D-Wave strategy dynamically detects dependencies and automatically exploits the parallelism if B frames are not used as reference frames.

MB-level parallelism has been exploited in previous work. Van der Tol et al. [6] presented the exploitation of intra-frame MB-level parallelism and suggested to

combine it with frame-level parallelism. If frame-level parallelism can be exploited is determined statically by the length of the motion vectors, while in our approach it is determined dynamically.

Chen et al. [5] also presented MB-level parallelism combined with frame-level parallelism to parallelize H.264 encoding. In their work, however, the exploitation of frame-level parallelism is limited to two consecutive frames and independent MBs are identified statically. This requires that the encoder limits the motion vector length. The scalability of the implementation is analyzed on a quad-core processor with Hyper-Threading Technology. In our work independent MBs are identified dynamically and we present results for up to 64 cores.

This paper is organized as follows. Section 2 provides an overview of MB parallelization technique for H.264 video decoding and the 3D-Wave technique. Section 3 presents the simulation environment and the experimental methodology to evaluate the 3D-Wave implementation. In Section 4 the implementation of the 3D-Wave on the embedded many-core is detailed and it introduces a frame scheduling policy to limit the number of frames in flight and describes a priority policy to reduce latency. The results of the 3D-Wave, the frame scheduling and frame priority policies are presented in Section 5. Conclusions are drawn in Section 6.

2 Thread-Level Parallelism in H.264 Video Decoding

Currently, one of the best video coding standard, in terms of compression and quality is H.264 [7]. The coding efficiency gains of advanced video codecs such as H.264 come at the price of increased computational requirements. The demands for computing power increases also with the shift towards high definition resolutions. As a result, current high performance uniprocessor architectures are not capable of providing the required performance for real-time processing [8,9]. Therefore, it is necessary to exploit parallelism. The H.264 codec can be parallelized either by a task-level or data-level decomposition.

In a *task-level decomposition* the functional partitions of the algorithm are assigned to different processors. Scalability is a problem because it is limited to the number of tasks, which typically is small. In a *data-level decomposition* the work (data) is divided into smaller parts and each part is assigned to a different processor. Each processor runs the same program but on different (multiple) data elements (SPMD). In H.264 data decomposition can be applied to different levels of the data structure. Due to space limitations only MB-level parallelism is described in this work. A discussion of the other levels can be found in [3].

In H.264, the motion vector prediction, intra prediction, and the deblocking filter kernels use data from neighboring MBs defining a set of dependencies shown as arrows in Figure 1. Processing MBs in a diagonal wavefront manner satisfies all the dependencies and allows to exploit parallelism between MBs. We refer to this parallelization technique as 2D-Wave, to distinguish it from the 3D-Wave proposed in [3] and for which implementation results are presented in this work.

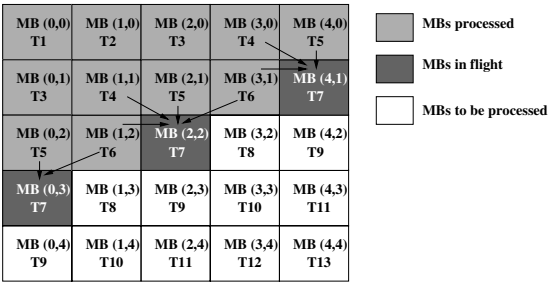


Fig. 1. 2D-Wave approach for exploiting MB parallelism. The arrows indicate dependencies.

Figure 1 illustrates the 2D-Wave for a 5×5 MBs image (80×80 pixels). At time slot T7 three independent MBs can be processed: MB (4,1), MB (2,2) and MB (0,3). The number of independent MBs varies over time. At the start of decoding a frame it increases with one MB every two time slots, then stabilizes at its maximum, and finally decreases at the same rate it increased. For a low resolution like QCIF there are at most 6 independent MBs during 4 time slots. For Full High Definition (1920x1088) there are at most 60 independent MBs during 9 time slots.

MB-level parallelism has several advantages over other H.264 parallelization schemes. First, this scheme can have a good scalability. As shown before the number of independent MBs increases with the resolution of the image. Second, it is possible to achieve a good load balancing if dynamic scheduling is used.

However, MB-level parallelism has some disadvantages. The first one is that the entropy decoding cannot be parallelized using data decomposition, due to the fact that the lowest level of data that can be parsed from the bitstream are slices. Only after entropy decoding has been performed the parallel processing of MBs can start. This disadvantage can be overcome by using special purpose instructions or hardware accelerators for entropy decoding. The second disadvantage is that the number of independent MBs is low at the start and at the end of decoding a frame. Therefore, it is not possible to sustain a certain processing rate during the decoding of a frame.

None of the approaches described scales to future many-core architectures containing 100 cores or more, unless extremely high resolution frames are used. We have proposed [3] a parallelization strategy that combines MB-level with frame-level parallelism and which reveals the large amount of parallelism required to effectively use future many-core CMPs. The key points are described below.

In the decoding process the dependency between frames is in the Motion Compensation (MC) module only. When the reference area has been decoded, it can be used by the referencing frame. Thus it is not necessary to wait until a frame is completely decoded before decoding the next frame. The decoding process of the next frame can be started after the reference areas of the reference frames are decoded. Figure 2 illustrates this way of parallel decoding of frames, called 3D-Wave strategy.

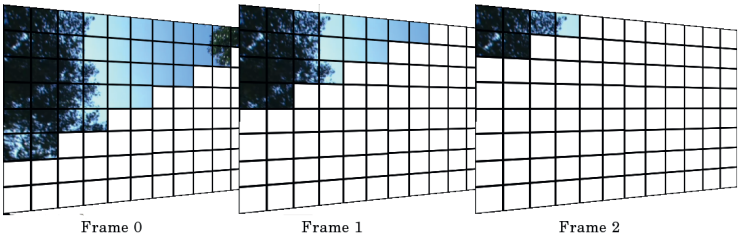


Fig. 2. 3D-Wave strategy: frames can be decoded in parallel because inter frame dependencies have limited spatial range

In our previous study the H.264 decoder was modified to analyze the available parallelism for real movies. The sequences of the HD-VideoBench [10] were used as inputs for the experiment. For each MB its dependencies were analyzed. Then for each timeslot we analyzed the number of MBs that can be processed in parallel during that time slot. The experiments did not consider any practical or implementation issues, but explored the limits to the parallelism available in the application.

The results show that the amount of MB-level parallelism exhibited by the 3D-Wave ranges from 1202 to 1944 MBs for SD resolution (720×576), from 2807 to 4579 MBs for HD resolution (1280×720), and from 4851 to 9169 MBs for FHD resolution (1920×1088). To sustain this amount of parallelism, the number of frames in flight ranges from 93 to 304 depending on the input sequence and the resolution.

The theoretically available parallelism exhibited by the 3D-Wave technique is huge. However, there are many factors in a real system, such as the memory hierarchy and bandwidth, that could limit its scalability. In the next sections the approach to implement the 3D-Wave and exploit this parallelism on an embedded multicore system is presented.

3 Experimental Methodology

In this section the tools and methodology to implement and evaluate the 3D-Wave technique are detailed. Also components of the many-core system simulator used to evaluate the technique are presented. An NXP proprietary simulator based on SystemC is used to run the application and collect performance data. Computations on the cores are modeled cycle-accurate. The memory system is modeled using average data transfer times with channel and bank contention detection that modifies the latency to transfer the data. The simulator is capable of simulating systems with 1 to 64 TM3270 cores with shared memory and its cache coherence protocols. The simulator does not simulate the operating system.

The TM3270 [11] is a VLIW-based media-processor based on the Trimedia architecture. It addresses the requirements of multi-standard video processing at standard resolution and the associated audio processing requirements for the consumer market. The architecture supports VLIW instructions with five issue

slots. Each one is guarded. The pipeline depth varies from 7 to 12 stages. Address and data words are 32 bits wide. It features a unified register file with 128 32-bit registers. The SIMD capabilities are 2 x 16-bit and 4 x 8-bit. The 64Kbyte data-cache has 64-byte lines and is 4-way set-associative with LRU replacement and write allocate. The instruction cache is not modeled. The TM3270 processor can run at up to 350 MHz, but in this work the clock frequency is set to 300 MHz. To produce code for the TM3270 the state-of-the-art highly optimizing NXP TriMedia C/C++ compiler version 5.1 is used.

The modeled system features a shared memory using MESI cache-coherence protocol. Each core has its own L1 data cache and can copy data from other L1 caches through 4 channels. The cores share a distributed L2 cache with 8 banks and an average access time of 40 cycles. The average access time takes into account L2 hits, misses, and interconnect delays. L2 bank contention is modeled so two cores cannot access the same bank simultaneously.

The multi-core programming model follows the task pool model. A Task Pool (TP) library implements submissions and requests of tasks to/from the task pool, synchronization mechanisms, and the task pool itself. In this model there is one main core and the other cores of the system act as slaves. Each slave runs a thread by requesting a task from the TP, executing it, and requesting another task. This allows low task execution overhead of less than 2% of the average MB decoding time for task request.

The experiments focus on the baseline profile of the H.264 standard. The baseline profile only supports I and P frames and every frame can be used as a reference frame. This feature prevents the exploitation of frame-level parallelization techniques such as the one described in [5]. However, this profile highlights the advantages of the 3D-Wave. In this profile, the scalability gains come purely from the application of the 3D-Wave technique. Encoding was done with the X264 encoder [12] using the following options: no B-frames, maximally 16 reference frames, weighted prediction, hexagonal motion estimation algorithm with maximum search range 24, and one slice per frame. The experiments use all four videos from the HD-VideoBench [10], Blue_Sky, Rush_Hour, Pedestrian, and Riverbed, in the three available resolutions, SD, HD and FHD.

The 3D-Wave technique focuses on the thread-level parallelism available in the MB processing kernels of the decoder. The Entropy decoder is known to be difficult to parallelize. To avoid the influence of the entropy decoder, its output has been buffered and its decoding time is not taken into account. Although not the main target, the 3D-Wave also eases the entropy decoding challenge. Since entropy decoding dependencies do not cross slice/frame borders, multiple entropy decoders can be used.

4 Implementation

In our previous work we used the FFmpeg decoder, but since we are using the Trimedia simulator for this implementation, we use the NXP H.264 decoder. The 2D-Wave parallelization strategy has already been implemented in this

decoder [13], making it a perfect starting point for the implementation of the 3D-Wave. The NXP H.264 decoder is highly optimized, including both machine-dependent optimizations (e.g. SIMD operations) and machine-independent optimizations (e.g. code restructuring).

The 3D-Wave implementation serves as a proof of concept thus the implementation of all features of H.264 is not necessary. Intra prediction inputs are deblock filtered samples instead of unfiltered samples as specified in the standard. However, this does not add visual artifacts to the decoded frames or change the MB dependencies.

This section details the 2D-Wave implementation used as the starting point, the 3D-Wave implementation, and the frame scheduling and priority policies.

4.1 2D-Wave Decoder

The MB processing tasks are divided in four kernels: vector prediction (VP), picture prediction (PP), deblocking info (DI), and deblocking filter (DF). VP calculates the motion vectors (MVs) based on the predicted motion vectors of the neighbor MBs and the differential motion vector present in the bitstream. PP performs the reconstruction of the MB based on neighboring pixel information (Intra Prediction) or on reference frame areas (Motion Compensation). Inverse quantization and inverse transform are also part of this kernel. DI calculates the strength of the DF based on MB data, such as the MBs type and MVs. DF smoothes block edges to reduce blocking artifacts.

The 2D-Wave is implemented per kernel. By this we mean that first VP is performed for all MBs in a frame, then PP for all MBs, etc. Each kernel is parallelized as follows. Figure 1 shows that each MB depends on at most four MBs. These dependencies are covered by the dependencies from the left MB to the current MB and from the upper right MB to the current MB, i.e., if these dependencies are satisfied then all dependencies are satisfied. Therefore, each MB is associated with a reference count between 0 and 2 representing the number of MBs on which it depends. When a MB is finished, the reference counts of the MBs that depend on it are decreased. When one of these counts reaches zero, a thread that will process the associated MB is submitted to the TP.

When a core loads a MB in its cache, it also fetches neighboring MBs. Therefore, locality can be improved if the same core also processes the right MB. To increase locality and reduce task submission and acquisition overhead, the 2D-Wave implementation features an optimization called *tail submit*. After the MB is processed, the reference counts of the MB candidates are checked. If both MB candidates are ready to execute, the core processes the right MB and submits the other one to the task pool. If only one MB is ready, the core starts its processing without submitting or acquiring tasks to/from the TP. In case there is no neighboring MB ready to be processed, the task finishes and the core request another one from the TP. Figure 3 depicts pseudo-code for MB decoding after the tail submit optimization has been performed.

```
void deblock_mb(int x, int y){
again:
   // ... the actual work

   ready1 = x>=1 && y!=h-1 && atomic_dec(&deblock_ready[x-1][y+1])==0;
   ready2 = x!=w-1 && atomic_dec(&deblock_ready[x+1][y])==0;

   if (ready1 && ready2){
      tp_submit(deblock_mb, x-1, y+1);    // submit left-down block
      x++;
      goto again;                         // goto right block
   }
   else if (ready1){
      x--; y++;
      goto again;                         // goto left-down block
   }
   else if (ready2){
      x++;
      goto again;                         // goto right block
   }
}
```

Fig. 3. Tail submit

4.2 3D-Wave Implementation

In this section the 3D-Wave implementation is described. First we note that the original structure of the decoder is not suitable for the 3D-Wave strategy, because inter-frame dependencies are satisfied only after the DF is applied. To implement the 3D-Wave, it is necessary to develop a version in which the kernels are applied on a MB basis rather than on a slice/frame basis. In other words, we have a function `decode_mb` that applies each kernel to a MB.

In the 3D-Wave implementation multiple frames are decoded concurrently which requires modifications to the Reference Frame Buffer (RFB). The RFB stores the decoded frames that are going to be used as reference. As it can service only one frame in flight, the 3D-Wave would require multiple RFBs. Instead, in this proof of concept implementation, the RFB was modified such that a single instance can service all frames in flight. In the new RFB all the decoded frames are stored. The mapping of the reference frame index to RFB index was changed accordingly.

Figure 4 depicts pseudo-code for the `decode_mb` function. It relies on the ability to test if the reference MBs (RMBs) of the current MB have already been decoded or not. The RMB is defined as the MB in the bottom right corner of the reference area, including the extra samples for fractional motion compensation. To be able to test this, first the RMBs have to be calculated. If an RMB has not been processed yet, a method is needed to resume the execution of this MB after the RMB is ready.

```
void decode_mb(int x, int y, int skip, int RMB_start){
    IF !skip {
        Vector_Prediction(x,y);
        RMB_List = RMB_Calculation(x,y);
    }
    FOR RMB_start TO RMB_List.last{
        IF !RMB.Ready {
            RMB.Subscribe(x, y);
            return;
        }
    }
    Picture_Prediction(x,y);
    Deblocking_Info(x,y);
    Deblocking_Filter(x,y);
    Ready[x][y] = true;
    FOR KoL.start TO KoL.last tp_submit(MB);
    //TAIL_SUBMIT
}
```

Fig. 4. Pseudo-code for 3D-Wave

The RMBs can only be calculated after motion vector prediction, which also defines the reference frames. Each MB can be partitioned in up to four 8x8 pixel areas and each one of them can be partitioned in up to four 4x4 pixel blocks The 4x4 blocks in an 8x8 partition share the reference frame. With the MVs and reference frames information, it is possible to calculate the RMB of each MB partition. This is done by adding the MV, the size of the partition, the position of the current MB, and the additional area for fractional motion compensation and by dividing the result by 16, the size of the MB. The RMB results of each partition is added to a list associated with the MB data structure, called the RMB-list. To reduce the number of RMBs to be tested, the reference frame of each RMB is checked. If two RMBs are in the same reference frame, only the one with the larger 2D-Wave decoding order (see Figure 1) is added to the list.

The first time `decode_mb` is called for a specific MB it is called with the parameter `skip` set to `false` and `RMB_start` set to `0`. If the decoding of this MB is resumed, it is called with the parameter `skip` set to `true`. Also `RMB_start` carries the position of the MB in the RMB-list to be tested next.

Once the RMB-list of the current MB is computed, it is verified if each RMB in the list has already been decoded or not. Each frame is associated with a MB ready matrix, similar to the `deblock_ready` matrix in Figure 3. The corresponding MB position in the ready matrix associated with the reference frame is atomically checked. If all RMBs are decoded, the decoding of this MB can continue.

To handle the cases where a RMB is not ready, a RMB subscription technique has been developed. The technique was motivated by the specifics of the TP library, such as low thread creation overhead and no sleep/wake up capabilities. Each MB data structure has a second list called the Kick-off List (KoL) which

contains the parameters of the MBs subscribed to this RMB. When an RMB test fails, the current MB subscribes itself to the KoL of the RMB and finishes its execution. Each MB, after finishing its processing, indicates that it is ready in the ready matrix and verifies its KoL. A new task is submitted to the TP for each MB in the KoL.

The subscription process is repeated until all RMBs are ready. Finally, the intra-frame MBs that depend on this MB are submitted to the TP using tail submit, identical to Figure 3.

4.3 Frame Scheduling Policies

To achieve the highest speedup, all frames of the sequence are scheduled to run as soon as their dependencies are met. However, this can lead to a large number of frames in flight and large memory requirements, since every frame must be kept in memory. Mostly it is not necessary to decode a frame as soon as possible to keep all cores busy. A frame scheduling technique was developed to keep the working set to its minimum.

Frame scheduling uses the RMB subscription mechanism to define the moment when the processing of the next frame should be started. The first MB of the next frame can be subscribed to start after a specific MB of the current frame. With this simple mechanism it is possible to control the number of frames in flight. Adjusting the number of frames in flight is done by selecting an earlier or later MB with which the first MB of the next frame will be subscribed.

4.4 Task Priorities

In video decoding, latency is an important characteristic of the system. The frame scheduling policy described in the previous section reduces the frame latency. However, as a new frame is scheduled to be decoded, the available cores are distributed equally among the frames in flight. A priority mechanism was added to the TP library in order to reduce the frame decoding latency.

The TP library was modified to support two levels of priority. An extra task buffer was implemented to store high priority tasks. When the TP receives a task request, it first checks if there is a task in the high priority buffer. If so this task is selected, otherwise a task in the low priority buffer is selected. With this simple mechanism it is possible to give priority to the tasks belonging to the frame "next in line". Before submitting a new task the process checks if its frame is the frame "next in line". If so the task is submitted with high priority. Otherwise the submission to the TP is made using the low priority. This mechanism does not lead to starvation because if there is not sufficient parallelism in the frame "next in line" the low priority tasks are selected.

5 Experimental Results

In this section the experimental results are presented. The results include the scalability results of the 3D-Wave, the impact on the memory and bandwidth

requirements, and the results of the frame scheduling and priority policies. The experiments were carried out according to the methodology described in Section 3. To evaluate the 3D-Wave technique, one second (25 frames) of each sequence was decoded using the enhanced NXP decoder. Due to long simulation times and the large number of simulations, more frames could not be simulated. The four sequences of the HD-VideoBench using three resolutions were evaluated. Due to space limitations only the results for the Rush_Hour sequence are presented which are close to the average. The results for the other sequences vary less than 5%.

5.1 Scalability

The scalability results are for 1 to 64 cores. More cores could not be simulated due to limitations of the simulator. Table 1 depicts the scalability results, i.e., the speedup of the parallel implementation running on p processors over the parallel implementation running on a single core, of the 2D-Wave (columns labeled 2D-W) and 3D-Wave (columns labeled 3D-W) implementations. In addition, it shows the speedup of the 3D-Wave running on p cores over the 2D-Wave on p cores (columns labeled 3D vs 2D). On a single core, 2D-Wave can decode 39 SD, 18 HD, and 8 FHD frames per second, respectively.

Table 1. 2D-Wave and 3D-Wave speedups for the 25-frame Rush Hour sequences

Cores	SD			HD			FHD		
	2D-W	3D-W	3D vs 2D	2D-W	3D-W	3D vs 2D	2D-W	3D-W	3D vs 2D
1	1.00	1.00	0.92	1.00	1.00	0.92	1.00	1.00	0.92
2	1.77	2.00	1.05	1.77	2.00	1.04	1.78	2.00	1.04
4	3.22	4.00	1.14	3.27	3.99	1.13	3.31	4.00	1.11
8	5.56	7.80	1.29	5.78	7.83	1.25	5.96	7.88	1.22
16	8.19	14.63	1.65	9.31	14.75	1.46	9.92	15.21	1.41
32	8.42	27.78	3.04	11.78	28.44	2.22	14.40	28.94	1.85
64	8.32	49.32	5.47	11.53	53.16	4.25	15.35	54.78	3.28

On a single core the 3D-Wave implementation takes 8% more time than the 2D-Wave implementation due to administrative overhead. The 3D-Wave implementation scales almost perfectly up to 8 cores, while the 2D-Wave implementation incurs a 11% efficiency drop even for 2 cores due to the following reason. The tail submit optimization assigns MBs to cores per line. At the end of a frame, when a core finishes its line and there is no other line to be decoded, in the 2D-Wave it remains idle until all cores have finished their line. If the last line happens to be slow the other cores wait for a long time and the core utilization is low. In the 3D-Wave, cores that finish their line, while there is no new line to be decoded, will be assigned a line of the next frame. Therefore, the core utilization as well as the scalability efficiency of the 3D-Wave is higher.

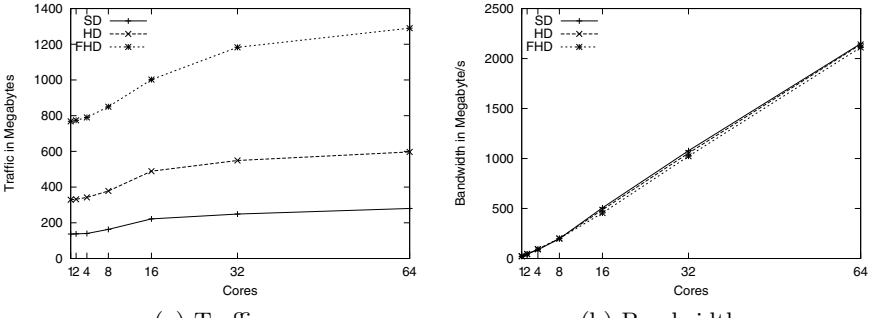
For SD sequences the 2D-Wave technique saturates at a speedup of just over 8 for 16 cores and beyond. This happens because of the limited amount of MB parallelism inside the frame and the dominant ramp up and ramp down of the availability of parallel MBs. The 3D-Wave technique for the same resolution continuously scales up to 64 cores with a parallelization efficiency just below 80%. For the FHD sequence, the saturation of the 2D-Wave occurs at 32 cores while the 3D-Wave continuously scales up to 64 cores with a parallelization efficiency of 85%.

The scalability results of the 3D-Wave implementation in Table 1 just slightly increase for higher resolutions. The 2D-Wave implementation on the other hand, achieves higher speedups for higher resolutions since the MB-level parallelism inside a frame increases. However, it would take an extremely large resolution for the 2D-Wave to leverage 64 cores, and the 3D-Wave implementation would still be more efficient.

The drop in scalability efficiency of the 3D-Wave for larger number of cores has two reasons. First, for large number of cores cache trashing occurs, as will be shown in the next section, which results in a large number of memory stalls. Second, at the start and at the end of a sequence, not all cores can be used because little parallelism is available. The more cores are used the more cycles are wasted during these two periods. In a real sequence with many frames it would be negligible.

5.2 Bandwidth Requirements

The impact of the 3D-Wave technique on memory and communication bandwidth has also been analyzed. First the data traffic between the L1 and L2 data caches is measured. Figure 5(a) depicts the traffic for the three resolutions. The graph shows that the 3D-Wave increases the data traffic by approximately 104%, 82%, and 68% when going from 1 to 64 cores, for SD, HD, and FHD, respectively. This increase in traffic is the result of cache thrashing. Data locality decreases as the number of cores increases, because the task scheduler does not take into account data locality when assigning a task to a core (except with the tail



(a) Traffic (b) Bandwidth

Fig. 5. Traffic and bandwidth for FHD Rush_Hour sequence

submit strategy). However, because 3D-Wave exploits inter-frame data locality, it results in an average 18% less traffic than 2D-Wave.

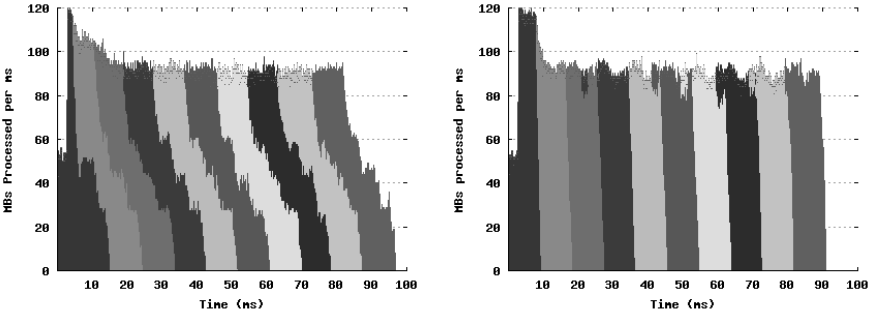
With the data traffic results it is possible to calculate the L2 to L1 bandwidth requirements. The bandwidth is calculated by dividing the total traffic by the time to decode the sequence in seconds. Figure 5(b) depicts the bandwidth results for different numbers of cores.

The total amount of intra chip bandwidth required for 64 cores is 2.1 GB/s for all resolutions of Rush_Hour sequence. The bandwidth is independent of the resolution because the number of MBs decoded per time unit per core is the same. This represents a workload more than 16 times higher than required for real time decoding, but it indicates what can be necessary for future applications such as 3D TV.

5.3 Frame Scheduling

Figure 6(a) presents the results of the frame scheduling technique applied to the FHD Rush_Hour sequence using a 16-core system. This figure presents the number of MBs processed per *ms*. It also shows to which frame these MBs belong. In this particular case the subscribe MB chosen is the last MB on the line that is at 1/3rd of the frame. For this configuration there are 3 frames in flight while there is a small performance loss of about 5%. This performance loss can be explained by the short sequence used. In these short sequences the time of ramp up and ramp down has a non-negligible impact on the overall performance.

In the current state of development, the selection of the subscribe MB must be done statically by the programmer. A methodology to dynamically fire new frames based on core utilization needs to be developed.



(a) Number of MBs processed per ms using frame scheduling and frames to which these MBs belong.

(b) Number of MBs processed per ms using frame scheduling and the priority policy.

Fig. 6. Results for frame scheduling and priority policy for FHD Rush_Hour in a 16-core processor. Different colors represent different frames.

5.4 Priority Policy

The priority mechanism, presented in Section 4.4, strongly reduces the latency of the frame to be decoded. In the original 3D-Wave implementation the latency of the first frame is 58.5 *ms*, using the FHD Rush_Hour sequence with 16 cores. Using the frame scheduling policy the latency drops to 15.1 *ms*. This latency is further reduced to 9.2 *ms* when the priority policy is applied together with frame scheduling. This is almost the same as the latency of the 2D-Wave, which decodes frames one-by-one. Figure 6(b) depicts the number of MBs processed per *ms* when this feature is used.

6 Conclusions

In this work an implementation of the 3D-Wave parallelization technique on an embedded CMP has been presented and evaluated. The implementation requires to identify intra-frame MB dependencies dynamically, which led to the development of a mechanism where MBs subscribe themselves to the MBs in the reference areas they depend upon. We have also presented policies for reducing the number of frames in flight and the frame latency.

The results show that the 3D-Wave implementation can leverage a multicore system with up to 64 cores. While the 2D-Wave has a speedup about 8 for 16 cores or more, for SD resolution, the 3D-Wave has a speedup of almost 45 on 64 cores. These results were achieved for sequences with no frame-, slice- or GOP-level parallelism.

Future work includes the development of an automatic frame scheduling technique, the implementation of the 3D-Wave on general purpose processors, and the implementation of the 3D-Wave in the encoder. A 3D-Wave implementation of the encoder can be applied for high definition, low latency encoding on multi-processors.

Acknowledgment

This work was partially supported by the European Commission in the context of the SARC integrated project #27648 (FP6), the Ministry of Science of Spain and European Union (FEDER funds) under contract TIC-2004-07739-C02-01, and the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). The authors would like to thank Anirban Lahiri from NXP for his collaboration on the experiments.

References

1. Okano, F., Kanazawa, M., Mitani, K., Hamasaki, K., Sugawara, M., Seino, M., Mochimaru, A., Doi, K.: Ultrahigh-Definition Television System With 4000 Scanning Lines. In: Proc. NAB Broadcast Engineering Conference, pp. 437–440 (2004)

2. Drose, M., Clemens, C., Sikora, T.: Extending Single-View Scalable Video Coding to Multi-View Based on H. 264/AVC. In: IEEE Inter. Conf. on Proc. Image Processing, pp. 2977–2980 (2006)
3. Meenderinck, C., Azevedo, A., Alvarez, M., Juurlink, B., Ramirez, A.: Parallel Scalability of H.264. In: Proc. First Workshop on Programmability Issues for Multi-Core Computers (January 2008)
4. Rodriguez, A., Gonzalez, A., Malumbres, M.P.: Hierarchical Parallelization of an H.264/AVC Video Encoder. In: Proc. Int. Symp. on Parallel Computing in Electrical Engineering, pp. 363–368 (2006)
5. Chen, Y., Li, E., Zhou, X., Ge, S.: Implementation of H. 264 Encoder and Decoder on Personal Computers. Journal of Visual Communications and Image Representation 17 (2006)
6. van der Tol, E., Jaspers, E., Gelderblom, R.: Mapping of H.264 Decoding on a Multiprocessor Architecture. In: Proc. SPIE Conf. on Image and Video Communications and Processing (2003)
7. Oelbaum, T., Baroncini, V., Tan, T., Fenimore, C.: Subjective Quality Assessment of the Emerging AVC/H.264 Video Coding Standard. In: Proc. Inter. Broadcast Conference (IBC) (2004)
8. Alvarez, M., Salami, E., Ramirez, A., Valero, M.: A Performance Characterization of High Definition Digital Video Decoding using H.264/AVC. In: Proc. IEEE Int. Workload Characterization Symposium, pp. 24–33 (2005)
9. Ostermann, J., Bormans, J., List, P., Marpe, D., Narroschke, M., Pereira, F., Stockhammer, T., Wedi, T.: Video Coding with H.264/AVC: Tools, Performance, and Complexity. IEEE Circuits and Systems Magazine 4(1), 7–28 (2004)
10. Alvarez, M., Salami, E., Ramirez, A., Valero, M.: HD-VideoBench: A Benchmark for Evaluating High Definition Digital Video Applications. In: IEEE Int. Symp. on Workload Characterization (2007)
11. van de Waerdt, J., Vassiliadis, S., Das, S., Mirolo, S., Yen, C., Zhong, B., Basto, C., van Itegem, J., Amirtharaj, D., Kalra, K., et al.: The tm3270 media-processor. In: Proc. 38th Inter. Symp. on Microarchitecture (MICRO), pp. 331–342 (2005)
12. X264. A Free H.264/AVC Encoder, `http://developers.videolan.org/x264.html`
13. Hoogerbrugge, J., Terechko, A.: A Multithreaded Multicore System for Embedded Media Processing. Transactions on High-Performance Embedded Architectures and Compilers 4(2) (to Appear, 2009)

Author Index