Olivier Temam
Pen-Chung Yew
Binyu Zang (Eds.)

Advanced Parallel Processing Technologies

LNCS 6965

9th International Symposium, APPT 2011
Shanghai, China, September 2011
Proceedings

Springer

Lecture Notes in Computer Science 6965

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

Olivier Temam Pen-Chung Yew
Binyu Zang (Eds.)

Advanced Parallel Processing Technologies

9th International Symposium, APPT 2011
Shanghai, China, September 26-27, 2011
Proceedings

Springer

Volume Editors

Olivier Temam
INRIA Saclay, Bâtiment G
Parc Club Université
Rue Jean Rostand, 91893 Orsay Cedex, France
E-mail: olivier.temam@inria.fr

Pen-Chung Yew
University of Minnesota at Twin Cities
Department of Computer Science and Engineering
200 Union Street, SE
Minneapolis, MN 55455, USA
E-mail: yew@cs.umn.edu

Binyu Zang
Fudan University
Software Building
825 Zhangheng Road
Shanghai 200433, China
E-mail: byzang@fudan.edu.cn

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-24150-5 e-ISBN 978-3-642-24151-2
DOI 10.1007/978-3-642-24151-2
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011936470

CR Subject Classification (1998): C.2.4, H.2, H.3, K.8.1, J.1, H.5, I.7, D.4, D.1

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

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Welcome to the proceedings of APPT 2011, which was held in Shanghai! With the continuity of Moore's law in the multicore era and the emerging cloud computing, parallelism has been pervasively available almost everywhere, from traditional processor pipelines to large-scale clusters. This creates grand challenges to architectural and system designs, as well as to methods of programming these systems, which form the core theme of APPT 2011. The two-day technical program of APPT 2011 provided an excellent venue capturing the state of the art and practice in parallel architectures, parallel software and distributed and cloud computing.

This biennial event provides a forum for representing this community's research efforts and exchanging viewpoints. We would like to express our thanks to all colleagues who submitted papers and congratulate those whose papers were accepted. As an event that has taken place for 16 years, APPT aims at providing a high-quality program for all attendees. We accepted 13 papers out of 40 submissions, presenting an acceptance rate of 32.5%.

To ensure a high-quality program and ensure interactive discussions, we made authors aware of the existence of a pre-filtering mechanism. We read all submissions, filtered those without enough technical merit before passing them to the Program Committee (PC). In total, we rejected three papers in this round. Then, each remaining submission got reviewed by at least three PC members. Many submissions were reviewed by four or five PC members, which yield high-quality reviews for each submission. Finally, an online PC meeting was held during July 4–8, to reach consensus for each submission.

In addition to the authors, we would also like to show our sincere appreciation to this year's dream-team PC. The 36 PC members did an excellent job in returning high-quality reviews in time. This ensured timely delivery of the review results to all authors.

Finally, we would like to thank the efforts of our General Chair (Xuejun Yang), our conference coordinator (Yong Dou) and the support of our industry sponsor (Intel), the Special Interests Group of Computer Architecture in China Computer Federation, and National Laboratory for Parallel and Distributed Processing, China. Our thanks also goes to Springer for its assistance in putting the proceedings together. Their help made APPT 2011 a great success.

September 2011

Olivier Temam
Pen-chung Yew
Binyu Zang

Organization

General Chair

Xuejun Yang	NUDT, China

Program Co-chairs

Olivier Temam	INRIA, France
Pen-chung Yew	Academia Sinica and UMN, Taiwan and USA
Binyu Zang	Fudan University, China

Local Arrangements Chair

Enmei Tang	Fudan University, China

Publication Chair

Rudolf Fleischer	Fudan University, China

Financial Chair

Xiaotong Gao	Fudan University, China

Program Committee

Sanjeev Aggarwal	Indian Institute of Technology, Kanpur, India
David August	Princeton University, USA
John Cavazos	University of Delaware, USA
Wenguang Chen	Tsinghua University, China
Haibo Chen	Fudan University, China
Sangyeun Cho	University of Pittsburgh, USA
Robert Cohn	Intel, USA
Koen DeBosschere	Ghent University, Belgium
Yong Dou	National University of Defense Technology, China
Rudi Eigenmann	Purdue University, USA
Paolo Faraboschi	HP Labs
R. Govindarajan	Indian Institute of Science, Bangalore, India
Haibing Guan	Shanghai Jiaotong University, China
Yinhe Han	Chinese Academy of Sciences, China

Bo Huang	Intel, China
Axel Jantsch	KTH, Sweden
Keiji Kimura	Waseda University, Japan
Jingfei Kong	AMD, USA
Jaejin Lee	Seoul Nation University, Korea
Xiaofei Liao	Huazhong University of Science & Technology, China
Xiang Long	Beihang University, China
Zhonghai Lu	KTH, Sweden
Yinwei Luo	Peking University, China
Mike O'Boyle	University of Edinburgh, UK
Alex Ramirez	Barcelona Supercomputing Center, Spain
Lawrence Rauchwerger	Texas A&M University, USA
Vivek Sarkar	Rice University, USA
Per Stenstrom	Chalmers University, Sweden
Olivier Temam	INRIA, France
Dongsheng Wang	Tsinghua University, China
Jon Weissman	University of Minnesota, USA
Chengyong Wu	Chinese Academy of Sciences, China
Jan-Jan Wu	Academia Sinica, Taiwan
Pen-Chung Yew	Academia Sinica and UMN, Taiwan and USA
Binyu Zang	Fudan University, China
Antonia Zhai	University of Minnesota, USA

Table of Contents

Reconstructing Hardware Transactional Memory for Workload Optimized Systems*

Kunal Korgaonkar, Prabhat Jain, Deepak Tomar,
Kashyap Garimella, and Veezhinathan Kamakoti

RISE - Reconfigurable and Intelligent Systems Engineering Group
Indian Institute of Technology Madras, Chennai, India
kama@cse.iitm.ac.in

Abstract. Workload optimized systems consisting of large number of general and special purpose cores, and with a support for shared memory programming, are slowly becoming prevalent. One of the major impediments for effective parallel programming on these systems is lock-based synchronization. An alternate synchronization solution called Transactional Memory (TM) is currently being explored. We observe that most of the TM design proposals in literature are catered to match the constrains of general purpose computing platforms. Given the fact that workload optimized systems utilize wider hardware design spaces and on-chip parallelism, we argue that Hardware Transactional Memory (HTM) can be a suitable implementation choice for these systems. We re-evaluate the criteria to be satisfied by a HTM and identify possible scope for relaxations in the context of workload optimized systems. Based on the relaxed criteria, we demonstrate the scope for building HTM design variants, such that, each variant caters to a specific workload requirement. We carry out suitable experiments to bring about the trade-off between the design variants. Overall, we show how the knowledge about the workload is extremely useful to make appropriate design choices in the workload optimized HTM.

Keywords: Transactional Memory, Hardware Transactional Memory, Workload Optimized Systems.

1 Introduction

In this decade, besides adding more and more general purpose cores, processor vendors are also exploring the option of adding specialized cores to their newer architectures. At present, there is a keen interest shown by processor vendors towards *workload optimized systems* [1, 2]. Prominent in this league is IBM, whose recent offerings, namely, Wire Speed Processor, Network Intrusion Prevention System and Service Oriented Architecture Appliance [3] illustrate the emerging relevance of workload optimized systems in different application domains.

* The research was funded by Intel under the IITM-Intel Shared University Research program.

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 1–15, 2011.

The approach of designing the hardware for a workload optimized system is fundamentally different from that of the corresponding approach for a general purpose multi-core based system. General purpose computing involves usage of basic hardware blocks that are expected to suite the needs of diverse workloads. The economy of scale and power efficiency constraints force the use of simpler hardware. Only software-specified policy changes are possible. There is no opportunity to carry out hardware optimization (in terms of power or performance) even though there may be an ample scope for such an optimization in a given workload. On the contrary, the most basic and essential condition in the design of a workload optimized system is the presence of a wider hardware design space. A workload optimized system effectively exploits this wider hardware design space to achieve improved performance and power benefits for a specific workload.

A workload optimized system is usually built using a heterogeneous architecture that consists of a large number of general purpose cores coexisting with some special-purpose cores called accelerators. The multiple general purpose cores perform the task of running software threads, while the special purpose cores perform acceleration of key software components[1]. In terms of its programming interface, the accelerators can be programmed not just through system interfaces but also directly through user level code. However, to bring about some programming generality and also to reduce the overheads of explicit synchronization, workload optimized systems provide one single shared address space for normal cores as well as accelerators [1, 3]. It is important to note that the accelerators may have their own local caches and TLBs [3]. A shared address space enables usage of shared memory based parallel programming techniques.

1.1 Synchronization Bottleneck and TM as a Solution

The newer parallel architectures, both general purpose multi-core as well as heterogeneous architecture, have created an opportunity for various types of workloads from numerous application domains. Unfortunately, at present there is a scarcity of scalable software for these workloads. One of the major issues hindering the creation of scalable parallel software is *lock-based shared memory synchronization*. Lock-based synchronization is considered unsuitable for upcoming parallel architectures due to their failure to simultaneously provide good core utilization and manageable programming complexity.

Transactional memory (TM) [4] has been suggested by researchers as a possible solution. TM provides programmers with a simple interface using which critical sections of the code gets automatically transformed as atomic transactions. TM provides both performance scalability and high programmer productivity. Importantly, TM operates on the shared memory address space, and thus can be used on platforms supporting programming generality. TM can be implemented either in hardware (HTM [4]) or in software(STM [5]), or in a combination of both [6, 7].

[1] We categorize workload optimized systems under heterogeneous computing, and the conventional multi-core architectures with look-alike cores under general purpose computing.

1.2 Revisiting HTM for Workload Optimized Systems

While research in TM has been able to demonstrate its benefits both in terms of performance [4] and programmability [8], it has also surfaced key difficulties in building a scalable and completely-programmer-oblivious TM implementation. Despite the initial promise, there has been many unresolved issues and unsolved problems in implementing TM both as a STM as well as a HTM [9, 10]. This has lead to recent attempts to combine the best of STM and HTM as Hardware-Software TM designs, in the form of Hybrid TM [6] and Hardware Accelerated STM [7]. We acknowledge that attempts towards Hardware-Software TM designs do provide a viable path for eventual adoption of TM in general purpose computing. However, in this paper, we attempt to *revisit* HTM design in the context of workload optimized systems. Our actions are motivated by the following reasons.

1. Workload optimized systems provide a shared memory programming interface. Thus transactional memory, irrespective of the type of its implementation, is one of the strong candidates for synchronization on these systems.
2. A workload optimized system is usually designed for a specific workload in a bottom-up hardware-first manner. On the contrary, a Hardware-Software TM design employs limited hardware and is required to cater to the diverse workloads of a general purpose computing environment. Thus Hardware-Software TM design may not be the most ideal TM implementation for workload optimized systems.
3. In general workload optimized systems use *on-chip parallelism* to achieve maximum performance. HTM also uses on-chip parallelism, and can potentially provide higher performance than any other type of TM implementation. We therefore think that HTM shall hold higher relevance in workload optimized systems, as compared to other types of TM implementation.
4. We argue that a HTM catering to workload optimized systems should satisfy the demands of the workload rather than provide for complete TM semantics. We therefore think that there exist some scope for relaxing some of the TM criteria targeted by a HTM in the context of workload optimized systems.
5. A workload optimized system shall demand a wider hardware design space in a HTM so as to exercise different power-performance tradeoff. We think it is important to investigate whether current HTM designs in literature are alterable given various levels of performance requirements and area-power constraints.

1.3 Contributions of this Paper

1. We suggest possible relaxations to the criteria targeted by a HTM in the context of workload optimized systems (Section 1.4). We argue that the complete TM semantics hold more relevance, and quite rightly, only in general purpose computing. We also explain how the stated relaxations may not affect the usability of TM from the perspective of the end programmer.
2. We identify components and techniques from HTM literature that can possibly be used in a workload optimized HTM (Section 2). Unlike previous

HTM proposals, the illustrated workload optimized HTMs do not attempt to provide holistic and complete solutions.
3. We experimentally evaluate the standalone effectiveness as well as the relative performance benefits of the explored design options (Section 3). Our analysis of the results clearly shows how the knowledge of a workload can be used to make appropriate design choices in a workload optimized HTM.

1.4 Relaxation of the Criteria for a HTM

We argue that there is a scope for relaxation of the criteria targeted by a HTM in the context of workload optimized systems. We shall refer to the level of support in terms of size of a transaction as *size bound* of the HTM, and, the extent of interrupting events (for e.g. context switch) allowed and handled within a transaction as the *duration bound* of a HTM. For some workload optimized systems, we expect the required size or duration bound of the transactions to be known in advance. If not the exact bounds, we expect at least an approximate idea of the bounds to be known during design time. In this paper, we do not deal with workload optimized systems that have no any idea of the bounds or those which expect complete unboundedness. Existing advanced HTM designs [14] and Hardware-Software TM designs [6, 7] operate better in this area of the design space.

Apart from size and duration bound, performance per watt (and area) is also an important criteria from the perspective of workload optimized system. For a workload optimized system which sets a very high performance target, it is justifiable to have additional hardware to achieve the desired performance. On the other hand, for a workload optimized system which does not expect very high performance but has a very tight power budget, a simple low-cost rudimentary HTM may suffice. Finally, in terms of usability, we argue that relaxed criteria or semantics does not make the resultant HTM less programmer friendly. The programmer can freely use the complete TM semantics. The major change is in the role of the designer of the workload optimized system who has to rightly estimate the bounds, performance target and power budget of the required HTM.

2 Exploration of HTM Design Choices

Most of the existing HTM designs in literature were developed based on the understanding of the common-uncommon behaviour of standard benchmarks, coupled with the understanding of the ease of implementation and practicality of the individual hardware components. We think that in the context of workload optimized systems, the implementation and practicality of the individual hardware components in the design is highly workload dependent. Section 3 shall present the results of an empirical study on this argument. Therefore, while exploring HTM design choices for workload optimized system, we do not use the common-uncommon case approach. The following sections bring out the explored HTM design choices.

2.1 Base HTM Design Choices

We shall use figure 1 to project a sketch of the design choices, broadly categorized as base HTM design choices. For convenience, we shall call the overall HTM design used to demonstrate the design choices as base HTM design. The shaded portions in the figure are existing components of a heterogeneous architecture, while the un-shaded ones are newly added. We project three main design choices. This design is alterable and thus can be adapted to suit multiple categories of workloads depending on the *size* of their transactions. Please note that we do not claim the novelty for the individual HTM design choices. We only highlight the possibility of various choices. While building the base HTM design, besides the main choices, we also make other design choices which are required to create a working HTM (refer 'Other Design Considerations').

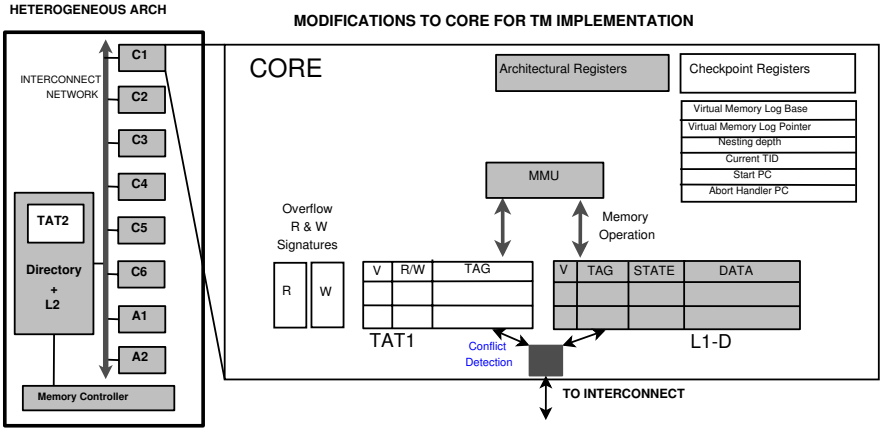


Fig. 1. A heretogeneous architecture with 6-cores (C1 to C6) and 2-Accelerators (A1, A2) with all cores and accelerators connected to an interconnect network. The accelerators may also have local caches [3] and the accompanying transactional hardware. TAT Entry Format: V bit - Valid/Invalid entry, TAG - Address accessed by transaction, R/W bit - Read/Write Entry, CID - Core Identifier (only for TAT2); Cache Entry Format: V bit - Valid/Invalid entry, TAG - Address accessed by normal memory operation, STATE - Cache Coherency State (e.g. MESI), DATA - Data Value.

Design Choice 1 - Transactional Access Table: The first design choice is use of a per-core structure called *Transactional Access Table* (TAT) used to handle the overflow of coherency permissions occurring from the per-core cache. We shall refer the per-core TAT structure as TAT1. A TAT1 entry[2] contains the address and the type (read/write) of operation. Please note, TAT1 stores the read and write addresses accessed by the currently running transaction, and not the data. Conflict detection is carried out against the cache as well as

[2] We shall use TAT entry or transactional entry interchangeably.

the TAT1 using the existing cache coherency protocol. We assume a directory coherency protocol, by means of which coherency messages are correctly sent to the appropriate cores.

TAT1 can be organized in a compact fashion using sector cache compression technique [11] (refer Figure 2). While a TAG of a sector stores the starting address of a memory location, all the individual bits of this sector shall map to consequent memory locations. The sector cache thus uses less storage area, but effectively tracks large number of transactional accesses. Design Choice 1 is majorly influenced by OneTM [12]. The experimental section (Section 3) gives a more detailed description on the subtleties with the achievable compression ratios.

Design Choice 2 - Overflow Signatures: We also explore the design option of using low-cost per-core signatures along with the per-core cache and the per-core TAT1. As per this design option, apart from cache and the TAT1, each core shall consists of a read and write overflow signatures to handle entries overflowing from TAT1. Design Choice 2 shall therefore provide improved levels of size bound than the Design Choice 1. By using signatures for overflow, Design Choice 2 gives away accuracy in return of reduced hardware cost and complexity. The size of the signature can be decided based on the requirement of the workload. We assume that the directory can keep track of the state of the addresses as present in the per-core coherency permission structures, including overflow signatures[3]. With regard to its use of signatures for overflow handling, the just described solution is similar to that by Shaogang Wang et al. [13][4].

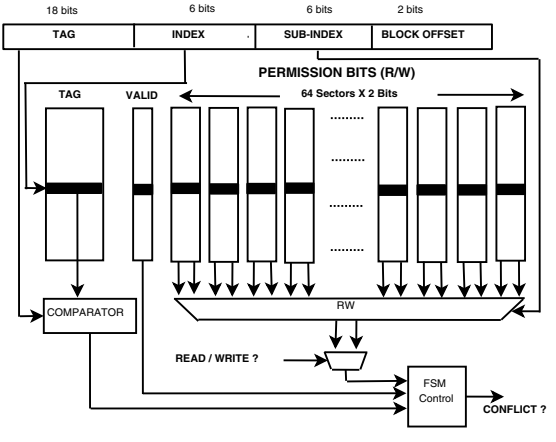


Fig. 2. Base HTM design: TAT

Design Choice 3 - TAT Hierarchy: Use of overflow signatures to handle TAT1 overflow is one design option. Next, we explore an alternative design option to provide higher levels of size bounds for workloads that demand such a feature. We explore on-chip mechanisms since they hold higher relevance in the context of workload optimized systems. We do not use a dedicated on-chip storage for overflow since that may incur very high additional hardware cost.

[3] In case of directory overflows, the overall state for a particular address can be reconstructed by sending a broadcast to all the cores.

[4] However, Shaogang Wang's solution did not use a compressed TAT structure.

Instead, we choose to *reuse* a section of the L2 cache to store transactional entries overflowing from any of the individual TAT1s. We do not adopt the technique of using meta-data bits derived from the error correction codes [12, 14] since they cannot represent all the transactional readers for a particular address. We instead explore the possibility of using some of the L2 data lines for storing transactional entries.

We call the section of the L2 cache used to store overflowing TAT1 entries as TAT2. In order to maintain an association between an overflowed TAT entry and its owner transaction, each TAT2 entry is augmented with Core Identifier field (CID). By using the large storage provided by TAT2 along with sector cache like compression, Design Choice 3 can provide improved levels of size bounds for hardware transactions. A 1 MB of storage in L2 can enable tracking of upto 256 times of transactional data (256 MB or 0.25 GB). This enables HTM for use by workloads that demand such a feature. Note this design option does *not* intend to, neither provides, complete size unboundedness.

The directory forwards the requests to the cores and conflict are detected against the coherency permission bits stored in TAT1 and the existing cache. Additionally, overflowing TAT1 entries are stored in TAT2. Conflict check can carried out against the TAT2 entries using an overflow tracking mechanism incorporated in the existing L2 pipeline. At the bare minimum such a mechanism needs a single transactional bit inside each L2 cache line, the rest of the meta-data is stored in the L2 data line. With some few additions, the existing L2 pipeline infrastructure [15] provides the basic operations to perform meta-data manipulation. The transactional entries belonging to the same transaction are linked together using pointers that are a part of the meta-data. To perform fast clean-up of TAT2 entries, we propose multiple low cost clean-up units that can perform overflow list traversal and clean up. We defer a more detailed design of TAT2 for future work.

Other Design Considerations: Irrespective of the choices made in terms of the improved size bounds, following are some of the other aspects of the base HTM design. Like LogTM [16], the design performs version management using per-thread virtual memory logs and simple software handlers. For handling thread context switching and thread migration, the OS completely defers these events while a transaction is active, or the transaction is simply aborted and restarted later. Conflict resolution is performed in software after the cache coherency identifies a conflict. The CIDs of the conflicting transactions are piggybacked by the directory.

2.2 Enhanced HTM Design Choices

In this section, we project one more design variant called enhanced HTM design. While base HTM design choices explored the aspect of supporting various sizes for transactions in a HTM, the enhanced HTM design examines the scope of handling OS events inside transactions, and also supporting multiple simultaneous transactional threads inside a core.

Transactional Access Table with TID Bits: Like the base HTM design, the enhanced HTM design uses a TAT based conflict detection mechanism. The TAT1 structure in the enhanced HTM design keeps track of entries belonging to multiple transactions. This is possible with the new organisation of the TAT1 structure. Extra vector of bits, called Transaction Identifier (TID) bits, is associated with each sector line. An entry belongs to a transaction if its corresponding bit is set in the vector. At any given time only one of the bits can be set (thus false sharing does not arise). The number of simultaneous transactions that can be supported in a core is limited by the number of bits provisioned in the vector. The organisation of the TID bits is similar to that proposed in [6]. The enhanced HTM implements version management using per-thread virtual memory logs and simple software handlers.

There is also the possibility of organizing the TAT like a N-way set-associative cache structure (with no data field), popularly known as transact cache [6, 13]. While the advantage of cache-like structure is that it enables n-way sharing for workloads that experiences high collision misses, the sector cache has the advantage of giving higher compression for workloads with high spatial locality. Section 3 gives more insight on this comparison.

TID Aware Coherency Protocol: In the enhanced design, the cache does not contain the TID bits, and is not involved in conflict detection. Only the TAT1 is used to carry out conflict detection. Unlike the base HTM design, the coherency permissions are stored in the TAT1 from the beginning, and not following the cache overflows. The directory keeps track of the presence of the coherency permissions in TAT1 neglecting the overflows occurring in the cache. In addition to this, the enhanced design suitably adapts the cache coherency protocol to make it TID aware. The TID information (or bits) are attached as payload of the normal cache coherency messages. The unique TID for a transaction is constructed by appending the TID bits with the core identifier field. The core identifier represents the core that runs the transaction. The reuse of the existing cache coherency ensures strong isolation support.

Improved Duration Bound and Support for MT Cores: Unlike [17, 18], the enhanced HTM design need not carry out any extra operation or processing when an OS event occurs inside a transaction. The enhanced design uses the following approach to provide for duration bound. The TAT1 uses the TID bits to keep track of the entries of previously running but uncompleted transactions (suspended transactions). The design option enables provisioning for a certain number of TID bits inside TAT1 depending on the requirements of the workload. The improved support for duration bound also implies implicit support for multi-threaded (MT) cores, wherein multiple hardware threads can now share the same TAT1 structure to store their respective transactional entries. The enhanced HTM design does not support thread migration.

Improvement in the duration bound may also necessitate a corresponding improvement in the size bounds of the transactions. We therefore suggest that

the enhanced design, like the base HTM, can reuse a section of the L2 cache to support larger size transactions. The TAT replacement policy ensures that *actively* accessed transactional entries, irrespective of the state of their threads (whether running or suspended), reside close to the cores. The most active entries will reside in TAT1, the less active in TAT2. The number of simultaneous hardware transactions are however restricted by the size of the provisioned TID vector. Also, the size of these transactions is restricted by the size of the TAT2 provisioned within the L2 cache. For workloads that demand complete unboundedness, we suggest available solutions in HTM and Hardware-Software TM literature.

3 Experimental Work

Having seen the scope for building custom HTM design variants for specific workload requirements, this section will validate the usefulness of these design variants. We shall also show how the performance of known HTM components and techniques is highly sensitive to the workload characteristics. Among the various possible design variants, following is a list of HTM design variants which we have considered for experimental evaluation.

1. **htm-transact-cache:** HTM design based on a separate decoupled cache-like structure, referred as transactional buffer or transact cache [6, 13]. This structure is used to implement TAT1. It contains only the TAG field (no data field) and the associativity can vary from 1 to n. The size of a transaction is limited by the size of this structure. There is no support for duration bound.
2. **htm-b-tat2 :** Base HTM design consisting of TAT1 and TAT2. TAT2 entry contains CID field. TAT1 organized as sector cache [12]. Support for higher size bound using TAT2, but no support for context switch and migration.
3. **htm-e-tat2 :** Enhanced HTM design consisting of TAT1 and TAT2. TAT1 contains TID bits. Support for context switching, but not for migration. TAT1 inside htm-e-tat2 is implemented either using sector cache - **e-tat2-sector-cache**, or n-way transact cache - **e-tat2-transact-cache**. For transact cache, each TAT line in a n-way set has an associated TID array.
4. **htm-sig:** HTM design based on normal and summary signatures [17, 18]. Transaction size is limited, and context switching done using summary signatures.

Following is the list of experiments we conducted to compare between design options, as well as to test the explored design options: (a) Scalability test of htm-b-tat2 design; (b) Comparison of duration bound capabilities of htm-e-tat2 against htm-sig; and (c) Comparative area and power analysis of sector cache based TAT1 against transact cache based TAT1.

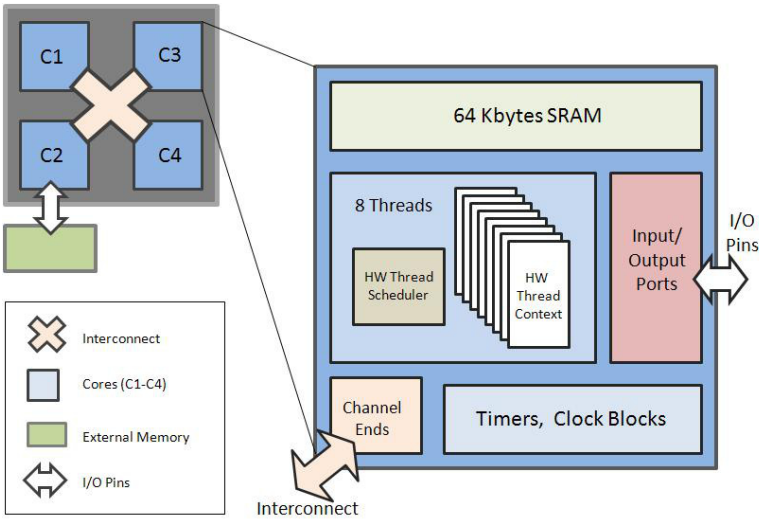


Fig. 3. XC-1 platform

3.1 Experimental Setup A

We implemented the design variants on the XMOS XC-1 platform [19] (Figure 3) using a novel hardware-software co-design approach. XC-1 is programmable using a C-like language called XC. The XC-1 contains a four-core processor. Each core has 8 hardware threads. All the threads can simultaneously run at a frequency of 100MHz. The transactional application, coded using XC, utilizes these hardware threads for execution. Some of these hardware threads were also used to implement the hardware components of the HTM design. A 64 KB memory block is available on each core which has a single cycle access latency. All the threads running on a core utilize this memory block for their stack, local data and the code. The XMOS setup lacks a common memory that can be shared across cores. Therefore, an external 32 MB SDRAM was attached to the platform. All the cores in XC-1 are connected through a programmable interconnect. XC-1 *channels* are used for inter-thread communication (inter-core as well as intra-core). The TID aware cache coherency was implemented using this interconnect.

Out of the four available cores, three of them, Core 0, Core 1 and Core 3, were used to run the transactional threads. TAT1 for each of these cores was stored in the local 64 KByte memory block of the core. Transactional threads performed transactional read and write operations on local TAT1. The normal memory hierarchy was also modelled on this platform. A part of the local 64 KByte memory block belonging to Core 0, Core 1 and Core 3 was used as L1 cache. The logic for the external memory controller was run on Core 2. External memory block was used to store L2 along with TAT2. The logic for implementing TAT2 was also run on Core 2. The interconnect was used by the transactional threads to communicate with the special threads running on Core 2.

Scalability Test of TAT Hierarchy: We executed the IntArray benchmark on htm-b-tat2 implementation on XC-1. The integer array was stored in the L2 cache. Threads randomly accessed data from the array of integers (of size 1000). Each transaction performed 1000 read or write operations with read write ratio = 80/20; and 10 such transactions were executed by each thread. After a transaction finished, the thread slept for some random time before starting the next transaction. The same benchmark was run with locks instead of transactional memory. Exponential back-off was used in case of transactions and locks to handle conflict and contention respectively. The results in Table 4 clearly show that htm-b-tat2 is much more efficient than locks.

Thread Count	4	8	12	16	20	24
Locks	140	251	541	893	1502	2185
htm-b-tat2	112	163	217	285	347	399

Fig. 4. Avg. execution time per critical section in μs

TID Bits versus Summary Signature: Since there were no standard transactional benchmarks dealing with OS events, we induced forceful context switch (*CS*) events in the IntArray benchmark so as to create a micro-benchmark. We used read-write set size as 1000 words, and array size as 10000. We observed that the thread count is the major metric for understanding the effectiveness of a duration bound capability of a HTM design. We consider two representative configurations: high per core thread count (24 threads) and low per core thread count (2 threads). The thread count includes both running as well as virtualized threads. Refer Table 1 and 2 for other parameters.

In htm-e-tat2, there is no direct cost (or processing cost) of a context switch. However, the design incurs an *indirect cost* (or collision cost). This is because, an existing TAT entry belonging to a thread may have to be evicted to TAT2 to create space for entries of other threads. There is also a cost associated with bringing back entries from TAT2 to TAT1 on being re-referenced. Table 3 shows the results. On one extreme, for sector cache based htm-e-tat2 with a large thread count, we observed a large overhead of 11.1 μs per context switch. On the other extreme, in case of 4-way transact cache with a low thread count, we observed the *lowest* overhead of 0.20 μs per context switch.

We also used the XC-1 platform to analyse performance of thread context switch support in some signature based designs. Frequently accessed data that was shared between cores was placed in the L2 cache. Results are shown in Table 3. In LogTM-SE [17], whenever a thread deschedules, it recalculates the summary signature, stores it in memory, and then synchronously notifies other threads by sending interrupts. The other threads on receiving the interrupt read the latest value of signatures from the memory. The above steps are performed each time compulsorily for any deschedule, reschedule or commit event. The overhead cost arises from the time taken to compute the summary signature and to carry out synchronous notification to *all* the other threads. We observed an enormous overhead of 521 μs for each of the above mentioned events. In LogTM-VSE [18], a lazy scheme is used for summary signature management.

Unlike [17], under the lazy scheme, summary signature is computed *only when required* and synchronous notifications are avoided. A deschedule or a commit event involves updating the local copy of the summary signature in the memory and a reschedule event involves computing the summary signature. The measured total per context switch overhead was only 6.24 μs. This overhead is *lowest* for high thread count compared to the corresponding overheads incurred by other design variants.

Overall summary of the results is as follows. In case of high thread count, e-tat2-sector-cache incurs a high overhead cost. One of the main reasons for this overhead is that all threads inside a core have to share a common sector cache unit. Also, since at any given time, only one thread can use a single TAT line, the occurrence of one thread throwing out an entry of another thread is very common. Thus, though a sector cache provides benefit in terms of supporting larger size transactions, it is not suitable for supporting workloads with multiple transactional threads inside a core. htm-e-tat2 therefore should not be preferred for workloads requiring efficient duration bound capability.

Table 1. TAT related details

Entries (Index)	64 (6 bits)
Sector Cache Bit Array	Pair of 64 bits
Word Size (Block Index)	4 bytes (2 bits)
Total TAT1 capacity	4096 words
TAT2 size	approx 1MB

Table 2. Other details

Context Switches per Tx.	10
Max TID bits (or threads)	24
Min TID bits (or threads)	2
Read write ratio	80/20
L1 cache line size	16 bytes
Signature size	2048 bits

Table 3. Avg. overhead per CS in μs; r-reschedule d-deschedule c-commit

Design	Overhead Cause (event in brackets)	24 threads/core (Overhead per CS)	2 threads/core (Overhead per CS)
TID bits based HTM (htm-e-tat2)			
sector-cache	collision cost	11.1	4.31
transact-cache (4 way)	collision cost	9.87	**0.20**
Summary Signature based HTM (htm-sig)			
logTM-SE [17]	computation & notification (r or d or c)	521	14.4
logTM-VSE [18]	update local copy (d or c)	**0.64**	0.64
	computation (r)	**5.60**	0.81
	total ((d or c) + (r))	**6.24**	1.45

A n-way transact cache with TID bits is more preferable option for duration bound since it can potentially reduce collision by enabling n-way sharing within the sets. By using a 4-way for a two thread configuration we observed the *least* overhead for duration bound as compared to the corresponding overheads incurred by the other design variants. Thus proving the ability of e-tat2-transact-cache to provide high performance duration bound support for workloads with low thread count. However, this advantage in e-tat2-transact-cache comes at the cost of increased hardware and low effective storage space for transactional entries (more on this in the next subsection).

The summary signature based designs use much lesser hardware and compensate by handling OS events by software handling. Further, design such as [18] uses a clever technique, called lazy scheme, to reduce the effective overhead involved in software handling. Our experimental results for [18] clearly indicates a low overhead in relative to the overheads incurred by the two htm-e-tat2 design variants. Thus, summary signature based design with clever software handling provide the *best* solution for workload optimized systems with large number of simultaneous transactional threads.

3.2 Experimental Setup B

We also developed appropriate RTL designs for the transact cache as well as for the sector cache based TAT1. Table 4 and Table 5 show the parameters of the implemented RTL designs. We used Synopsys Design Compiler to obtain a netlist from the RTL design, Synopsys Prime Power to calculate power, and Cadence Encounter to calculate the area. The specifications of the memory cells were obtained from Faraday's Memaker Memory Compiler. Following are some of the settings. For Primer Power: voltage 1.8 V, typical case temperature 25 °C, and clock frequency 33 MHz. For Memaker: TMSC's 180 nm CMOS process, and synchronous high speed single port RAM. Please note that due to certain limitations, we were able to acquire tools only for 180 nm technology. However, the projected results shall most likely show similar trends even for lower nanometer technology.

Sector Cache versus Transact Cache: The results of the area and power analysis are shown in Table 6. We obtain the power-consumption per unit entry as well as area-occupied per unit entry both for transact cache and the sector cache. Since these measures are normalized, it enables direct analyses and comparison. The results clearly indicate that compared to a transact cache, a sector cache is able to store more number of transactional entries within a given unit area. Sector cache enables a $4.6X$ times reduction in the effective area. However, the results also highlight the fact that sector cache is a suitable structure only if transactions exhibit high spatial locality of reference. The benefits of using sector cache shall diminish drastically if transactions do not exhibit good locality of reference. Another interesting observation was that though theoretically sector cache promises very high area-reduction ratios, in reality we did not achieve such reduction ratios. We attribute this to the memory blocks with large widths that are needed to realize sectors. Next, we observed as much as $2.77X$ reduction in power, which is again limited to transactions with high spatial locality of reference. Sadly, for transactions with low locality of reference there is a large penalty.

Overall the results for sector cache indicate that there is certainly a trade-off between attaining higher compression and managing the area and power budget. Targeting very high compression ratios using large-width memory blocks may have a big impact on the resultant area and the power of the TAT structure. Additionally, achievable compression ratio depends a lot on the workload's locality of reference properties. As far as transact cache is concerned, we saw

earlier that a n-way transact cache provides specific benefits in terms of duration bound for low thread count. However, the figures for a simple one-way transact cache clearly indicate high power-area cost. Thus both, the power-area plot of the transactional structure as well as the characteristic of the workload play a central role in deciding the right per-core transactional structure.

Table 4. 1-Way Transact Cache

TAG field	18 bits
Valid	1 bit
Read/Write	1 bit
Entries	1024

Table 5. Sector Cache

TAG field	18 bits
Valid	1 bit
Sector	Pair of 64 bits
Entries	64

Table 6. Performance Per Watt (PPW) and Performance Per Area (PPA). Transact Cache - 1024 entries; TAT - 4096 (64 lines, 64 sectors); Max/Max refer to Best/Worst locality of reference.

	Transact Cache		**Sector Cache**	
	Total	PPA/PPW	Total	PPA/PPW
Power	22.78 mW	22.24 μW	32.71 mW	**8.02** μW (Max) 511.09 μW (Min)
Area	0.52 mm^2	507.8 μm^2	0.45 mm^2	**109.86** μm^2 (Max) 7031.25 μm^2 (Min)

4 Conclusion

A signature based structure is most suited for conflict detection when the system is power-area constrained and the workload contains only small size transactions. Summary signature is most suited for workloads exhibiting high OS activity with many transactional threads. A n-way transact cache with TID bits is most suitable for workloads with few per-core transactional threads (to avoid overflow), more reads than writes and an overlapping access pattern (to maximize efficient utilization of transactional storage). Sector cache based TAT is most useful for workloads containing large size transactions but with high locality of reference. Since no structure seems to fulfill the demands of all types of workloads, it is important to at least understand the affinities of the currently available structures towards certain types of workloads. This understanding is necessary to build a truly workload optimized HTM.

References

[1] LaPotin, D.P., Daijavad, S., Johnson, C.L., Hunter, S.W., Ishizaki, K., Franke, H., Achilles, H.D., Dumarot, D.P., Greco, N.A., Davari, B.: Workload and Network-Optimized Computing Systems. IBM Journal of Res. and Dev. 54(1), 1:1–1:12 (2010)

[2] AMD APUs, http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx

[3] Franke, H., Xenidis, J., Basso, C., Bass, B.M., Woodward, S.S., Brown, J.D., Johnson, C.L.: Introduction to the Wire-Speed Processor and Architecture. IBM Journal of Res. and Dev. 54(1), 1–11 (2010)
[4] Herlihy, M., Eliot, J., Moss, B.: Transactional memory: Architectural Support for Lock-Free Data Structures. In: Proc. of the 20th Ann. Int. Symp. on Computer Architecture, pp. 289–300 (1993)
[5] Shavit, N., Touitou, D.: Software Transactional Memory. In: Proc. of the 14th Ann. ACM Symp. on Principles of Distributed Computing, pp. 204–213 (1995)
[6] Kumar, S., Chu, M., Hughes, C.J., Kundu, P., Nguyen, A.: Hybrid Transactional Memory. In: Proc. of Symp. on Principles and Practice of Parallel Programming, pp. 209–220 (2006)
[7] Shriraman, A., Marathe, V.J., Dwarkadas, S., Scott, M.L., Eisenstat, D., Heriot, C., Scherer, W.N., Michael, I., Spear, F.: Hardware Acceleration of Software Transactional Memory. Tech. Rep., Dept. of Computer Science, Univ. of Rochester (2006)
[8] Witchel, E., Rossbach, C.J., Hofmann, O.S.: Is Transactional Memory Programming Actually Easier? In: Proc. of the Principles and Practice of Parallel Programming, pp. 47–56 (2010)
[9] Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S., Chatterjee, S.: Software Transactional Memory: Why Is It Only a Research Toy? Queue 6(5), 46–58 (2008)
[10] McKenney, P.E., Michael, M.M., Walpole, J.: Why the Grass may not be Greener on the Other Side: A Comparison of Locking vs. Transactional Memory. In: Proc. of the 4th Workshop on Programming Languages and Operating Systems, pp. 1–5 (2007)
[11] Liptay, J.S.: Structural aspects of the System/360 Model 85, II: The Cache. IBM Sys. Journal 7(1), 15–21 (1968)
[12] Blundell, C., Devietti, J., Christopher Lewis, E., Martin, M.M.K.: Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. SIGARCH Comput. Archit. News 35(2), 24–34 (2007)
[13] Wang, S., Wu, D., Pang, Z., Yang, X.: Software Assisted Transact Cache to Support Efficient Unbounded Transactional Memory. In: Proc. of the 10th Int. Conf. on High Performance Computing and Communications, pp. 77–84 (2008)
[14] Bobba, J., Goyal, N., Hill, M.D., Swift, M.M., Wood, D.A.: TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In: Proc. of Int. Symp. on Computer Architecture, pp. 127–138 (2008)
[15] OpenSPARC T1, http://www.opensparc.net/opensparc-t1/index.html
[16] Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: Log-based Transactional Memory. In: Proc. of the 12th Int. Symp. on High-Performance Computer Architecture, pp. 254–265 (2006)
[17] Yen, L., Bobba, J., Marty, M.M., Moore, K.E., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A.: LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In: Proc. of the 13th Int. Symp. on High-Performance Computer Architecture, pp. 261–272 (2007)
[18] Swift, M., Volos, H., Goyal, N., Yen, L., Hill, M., Wood, D.: OS Support for Virtualizing Hardware Transactional Memory. In: Proc. of the 3rd Workshop on Transactional Computing (2008)
[19] Technology XMOS, http://www.xmos.com/technology

Enhanced Adaptive Insertion Policy for Shared Caches*

Chongmin Li, Dongsheng Wang, Yibo Xue, Haixia Wang, and Xi Zhang

Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science & Technology, Tsinghua University,
Beijing 100084, China
{lcm03,zhang-xi06}@mails.tsinghua.edu.cn,
{wds,yiboxue,hx-wang}@tsinghua.edu.cn

Abstract. The LRU replacement policy is commonly used in the last-level caches of multiprocessors. However, LRU policy does not work well for memory intensive workloads which working set are greater than the available cache size. When a new arrival cache block is inserted at the MRU position, it may never be reused until being evicted from the cache but occupy the cache space for a long time during its movement from the MRU to the LRU position. This results in inefficient use of cache space. If we insert a new cache block at the LRU position directly, the cache performance can be improved by keeping some fraction of the working sets is retained in the caches.

In this work, we propose Enhanced Dynamic Insertion Policy (EDIP) and Thread Aware Enhanced Dynamic Insertion Policy (TAEDIP) which can adjust the probability of insertion at MRU by set dueling. The run-time information of the previous and the next BIP level are gathered and compared with current level to choose an appropriate BIP level. At the same time, access frequency is used to choose a victim. In this way, our work can get less miss rate than LRU for workloads with large work set. For workloads with small working set, the miss rate of our design is close to LRU replacement policy. Simulation results in single core configuration with 1MB 16-way LLC show that EDIP reduces CPI over LRU and DIP by an average of 11.4% and 1.8% respectively. On quad-core configuration with 4MB 16-way LLC. TAEDIP improves the performance on the weighted speedup metric by 11.2% over LRU and 3.7% over TADIP on average. For fairness metric, TAEDIP improves the performance by 11.2% over LRU and 2.6% over TADIP on average.

1 Introduction

The Least Recently Used (LRU) replacement policy is commonly used in the on-chip memory hierarchies and is the *de-facto* standard of replacement policy.

* This work is supported by Nature Science Foundation of China under Grant No. 60833004,60970002, and the National 863 High Technology Research Program of China(No.2008AA01A201).

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 16–30, 2011.

In the LRU policy, a cache block must traverse from the MRU position to the LRU position before being evicted from the cache. The LRU policy can get high performance for workloads which working sets are smaller than the on-chip cache size or have high locality, however, LRU gets poor performance for memory intensive workloads which working sets are greater than the on-chip cache size such as the streaming workloads where most of the cache blocks are accessed only once. It is desirable that a replacement policy can perform well both for memory-intensive workloads and LRU-friendly workloads.

Numerous techniques have been proposed to improve the performance of LRU policy [9,3,4]. *Dynamic Insertion Policy* (DIP) [9] dynamically changes the insertion policy to get high performance for private caches with little hardware and design overhead. Bimodal Insertion Policy (BIP) [9] is thrashing-resistant by inserting the majority of the incoming cache block in the LRU position, only a few cache blocks are inserted at the MRU position. DIP dynamically chooses between the traditional LRU policy and BIP depending on which policy incurs fewer misses through runtime *Set Dueling* [10].

Thread-Aware Dynamic Insertion Policy (TADIP) [3] was proposed for multi-threaded workloads. TADIP chooses the insertion policy for each of the concurrently executing applications by their memory requirement through multi-set-dueling. TADIP proposed two scalable approaches without exponential increase in hardware overhead.

In this paper, we illustrate that the BIP policy with a fixed probability is not sufficient. We propose *Enhanced Dynamic Insertion Policy* (EDIP) which refines BIP into multiple levels, each level corresponds to a different probability that an incoming cache block would be inserted at the MRU position, from LIP (0%) to LRU (100%). EDIP uses the set dueling mechanism to gather runtime misses of the BIP at the pre-level and the next-level according the current BIP level. By adding multi-levels of insertion probability, EDIP can performance well for more complex access patterns.

We propose *Thread-Aware Enhanced Dynamic Insertion Policy* (TAEDIP) for shared caches, the set dueling mechanism in TADIP can be used for TAEDIP. We propose a new policy selection method for multiple threads which choice is made through comparison of three BIP levels.

Simulation results in single core configuration with 1MB 16-way LLC show that EDIP reduces CPI over LRU and DIP by an average of 11.4% and 1.8% respectively. On quad-core configuration with 4MB 16-way LLC. TAEDIP improves the performance on the weighted speedup metric by 11.2% over LRU and 3.7% over TADIP on average. For fairness metric, TAEDIP improves the performance by 11.2% over LRU and 2.6% over TADIP on average.

The rest of this paper is organized as follows, Section 2 provides background and motivation. Section 3 introduces EDIP and TAEDIP design for shared caches. Section 4 provides the experimental methodology and simulation results. Section 5 presents related work. This paper will be summarized in Section 6.

2 Background and Motivation

A cache replacement policy mainly concerns about two issues: the victim selection policy and the insertion policy. The victim selection policy determines which cache block to be evicted when a new data block needs to be accessed from a lower level memory hierarchy. The insertion policy decides where to place the incoming data block in the replacement list. Traditional LRU replacement policy inserts a new cache block at the MRU position of the replacement list to keep it on-chip for a long time. When an eviction is needed, the cache block in the LRU position is chosen as the victim. DIP[9]/TADIP [3] uses the same victim selection as LRU and inserts most of the incoming lines at the LRU position and the rest (with a low probability) lines are inserted at the MRU position.

Modern CMP typically use shared last-level caches to get high capacity utilization and avoid expensive off-chip accesses. When multiple applications compete for a shared LLC, it is a better choice to allocate more cache space for those applications that can benefit from cache and vice versa. However, traditional LRU is unable to tell whether an application can benefits from cache or not. LRU policy treats all incoming cache blocks uniformly. As a results, an application has no benefit from cache may occupy more cache than an application which benefits from cache. This suggests a cache management scheme that can allocate cache resources to applications based on their benefit from cache.

The access patterns of LLC can be classified into recency-friendly access patterns, thrashing access patterns, streaming access patterns and mixed access patterns [4]. LRU fits recency-friendly access patterns. BIP fits for thrashing and streaming access patterns. Mixed access patterns can be characterized as workloads where some references have a near re-reference interval while other references have a distant re-reference interval. Consequently, in the absence of scans, mixed access patterns prefer LRU. In the presence of streaming, LRU gets lower hit rate after the streaming. The optimal policy preserves the active working set in the cache after the stream completes. DIP [9] and TADIP [3] can reserve the active working set by keeping a little fraction of incoming data block at the MRU position.

DIP and TADIP assumed a cyclic reference model where an access pattern may repeat many times. However, the access stream visible/exposed to the LLC has filtered temporal locality due to the hits in the upper-level caches. The loss of temporal locality causes a significant percentage of L2 cache lines to remain unused or be reused only several times. When the size of references is greater than the available LLC size and the reuse time is small (such as two times), both LRU and BIP policies perform poor. Table 1 gives an example, for simplicity, we assume an access pattern $(a_0, a_1, ..., a_5)^2$ on a 4-way set. The cache blocks in the set are assigned for other data at the beginning. The accesses sequence and the blocks in the set are given in Table 1. Both the LRU and BIP(1/8) hit 0 time, while BIP(1/2) hits 2 times. BIP(1/8) performs poor in the previous case because it puts only 1/8 of the incoming lines at the LRU position. If we change probability from 1/8 to 1/2, the hit time is increased from 0 to 2. So if we can adjust the probability in BIP policy, more access patterns can benefit from the

Table 1. Hit count of different replacement policies

	Next	LRU	Hit count	BIP(1/8)	Hit count	BIP(1/2)	Hit count
1	a_0	$a_0x_xx_xx_x$	0	$x_xx_xx_xa_0$	0	$x_xx_xx_xa_0$	0
2	a_1	$a_1a_0x_xx_x$	0	$x_xx_xx_xa_1$	0	$a_1x_xx_xx_x$	0
3	a_2	$a_2a_1a_0x_x$	0	$x_xx_xx_xa_2$	0	$a_1x_xx_xa_2$	0
4	a_3	$a_3a_2a_1a_0$	0	$x_xx_xx_xa_3$	0	$a_3a_1x_xx_x$	0
5	a_4	$a_4a_3a_2a_1$	0	$x_xx_xx_xa_4$	0	$a_3a_1x_xa_4$	0
6	a_5	$a_5a_4a_3a_2$	0	$x_xx_xx_xa_5$	0	$a_5a_3a_1x_x$	0
7	a_0	$a_0a_5a_4a_3$	0	$x_xx_xx_xa_0$	0	$a_5a_3a_1a_0$	0
8	a_1	$a_1a_0a_5a_4$	0	$a_1x_xx_xx_x$	0	$a_1a_5a_3a_0$	1
9	a_2	$a_2a_1a_0a_5$	0	$a_1x_xx_xa_2$	0	$a_2a_1a_5a_3$	1
10	a_3	$a_3a_2a_1a_0$	0	$a_1x_xx_xa_3$	0	$a_3a_2a_1a_5$	2
11	a_4	$a_4a_3a_2a_1$	0	$a_1x_xx_xa_4$	0	$a_3a_2a_1a_4$	2
12	a_5	$a_5a_4a_3a_2$	0	$a_1x_xx_xa_5$	0	$a_5a_3a_2a_1$	2

replacement policy. The LRU policy is the fastest policy of evicting those x_x blocks from the cache set and BIP(1/8) policy is the slowest. BIP(1/2) finishes such eviction at the step-7. We name BIP with different probability as different BIP level. This enhanced dynamic insertion policy is described in detail in the next section.

3 Enhanced Adaptive Insertion Policy

3.1 Enhanced DIP for Single Core

In DIP [9], when BIP policy is chosen, it inserts an incoming cache block at the MRU position with a fixed very low probability. DIP assumed a cyclic access pattern which would repeat many times, however, this is not always true in the LLC because of filtered temporal locality. BIP with a fixed probability cannot adapt for all applications' behaviors. We refine the original BIP with changeable probability. Multiple BIP levels can fit for different access patterns.

In this work, we refine BIP into 8 levels, each level corresponds to a different probability that an incoming cache block will be inserted at the MRU position. Table 2 gives the probability of each level which is similar with the SPR replication level of ASR [1]. For Level-0, all incoming cache blocks are placed at the LRU position, which is the same as LIP policy, LRU corresponds to level-7. From Level-1 to Level-6, each level represents a BIP policy with a fixed probability of MRU insertion. By using fine grained BIP levels, we can change the insertion policy from LRU to LIP more accurately.

We use set dueling mechanism (SDM)[9] to gather the runtime miss rate of the two adjacent levels of current BIP level. For example, if the current BIP level is level-4, the pre-level and the next level need to monitor is level-3 and level-5 respectively. Fig. 1 depicts the positions of pre-level and next-level. All eight levels organized as a ring. When the current BIP level is 0, its pre-level and next

Table 2. Probability of different BIP insertion levels

Level	0	1	2	3	4	5	6	7
Probability	0	1/64	1/32	1/16	1/8	1/4	1/2	1
Threshold	0	2	4	8	16	32	64	128

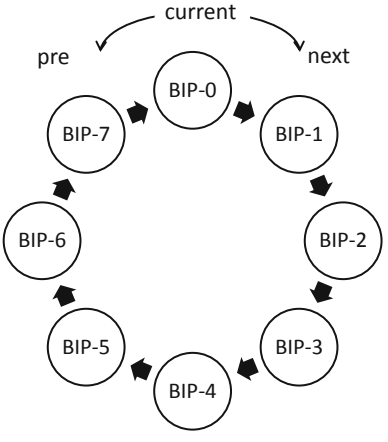


Fig. 1. The selection of pre-level and next-level

level are BIP-7 and BIP-1 respectively. For an arbitrary level, there are always two adjacent BIP levels used for comparison with the current level. In the set dueling of EDIP, two of each 32 cache sets are chosen as sampling sets, one for the pre-level and one for the next level, each has a Policy Selector (PSEL). A miss in the sets allocated for pre-level or next-level increments its PSEL. These two PSELs are decremented by the misses in the follower sets (current level). As the follower sets are much bigger than the size of watch sets (30 of each 32 sets), the global misses need to be scaled down before decreasing the PSEL. For a 16-way, 1MB LLC with 64B cache block, each SDM takes 32 sets, the follower sets should decrements the PSELs by one over every 30 misses. Fig. 2 illustrates the EDIP design. Compared with DIP, EDIP adds a PSEL over DIP, and needs to keep the value of current BIP-level (3-bits for 8 BIP levels), which overhead is very low.

When the SDM sets get more misses over the follower sets and one PSEL reaches a threshold, the current BIP level is changed to pre-level or next-level depending on which PSEL reaches the threshold. Then the two PSELs are reset as well as the value of pre-level and next-level of the SDMs.

3.2 Enhanced DIP for Shared Caches

Thread-Unaware Enhanced Dynamic Insertion Policy. When multiple applications compete for a shared LLC, it is a better choice to allocate more

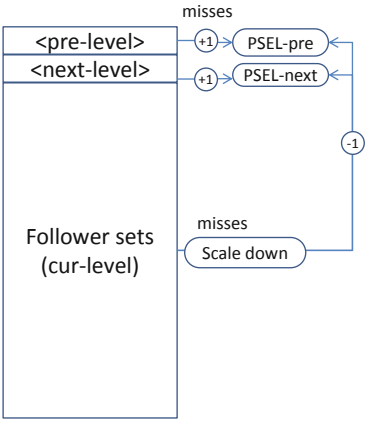


Fig. 2. EDIP via set dueling

cache space for those applications that can benefit from cache and vice versa. However, traditional LRU is unable to tell whether an application can benefit from cache or not. LRU policy treats all incoming cache blocks uniformly and ignores the changing of access pattern. As a results, an application has no benefit from cache capacity may received more cache than an application that benefit from cache capacity. This suggests a cache management scheme that can allocate cache resource to applications base on their benefit from cache.

EDIP can be directly applied in shared caches without modification as in single core. During the different application executing stages, which dynamically chooses an appropriate BIP level through set dueling and treats the requests from different threads as a whole and is unaware of different applications running at the system. TADIP [3] has proved that such thread-unaware scheme has low efficiency. We call this *Thread-Unaware Enhanced Adaptive Insertion* (TUEDIP). The design of TUEDIP is the same as EDIP. TUEDIP does not differentiate the behaviors of different applications, so TUEDIP is unable to distinguish between cache friendly and cache thrashing applications. As a result, EDIP applies a single policy for all applications for shared caches. If each application can be treated differently, such as using LRU policy to cache friendly applications and an appropriate BIP-level to applications that cause cache thrashing, the performance of the shared cache can be improved significantly. This motivates the need for a thread-aware insertion policy.

Thread-Aware Enhanced Adaptive Insertion. For the concurrently execution of multiple applications at the system, it is impractical to do a runtime comparison of all possible insertion policies. We propose *Thread-Aware Enhanced Adaptive Insertion* (TAEDIP) which tries to choose the appropriate BIP levels for each application. The decision is made by choosing the appropriate BIP level from pre-level, current-level and next-level for each application running at the system.

As TAEDIP needs to compare the cache performance among three BIP levels instead of two in TADIP, TAEDIP modifies the feedback mechanism as in TADIP-F [3]. TAEDIP requires 2N SDMs to learn the best insertion policy for each application given the insertion policies for all other applications. Of the 2N SDMs, a pair of SDMs are owned by every application. One of the SDMs in the pair is used for pre-level, and the other for next-level. For the owner application, the pre-SDM always uses the pre-level BIP policy and the next-SDM always uses the next-level BIP policy. For the remaining applications, the SDMs use the insertion policy which is currently used in the follower sets. For each application, the PSELs updating and BIP-level adjusting is similar to EDIP described above.

There are two schemes to update the PSELs of each application. In the first scheme, misses in the pre-SDM or next-SDM decrements the corresponding PSEL, misses in the follower sets increment all the PSELs (after scaling down). We name this scheme as *TAEDIP-Global* (TAEDIP-G). In the second scheme, each SDM only counts the misses of SDM owner's and the decrements the corresponding PSEL. Misses in the follower sets are also classified for different application(after scaling down) and increment the PSEL pair of the application, PSELs of other applications are unchanged. We name this scheme as *TAEDIP-Individual* (TAEDIP-I)

Fig. 3 and Fig. 4 illustrates TAEDIP for a cache shared by four applications. The cache has eight SDMs, one pair of SDMs for each application. For instance, the two SDMs for the first application use the tuple `<pre-0,P1,P2,P3>` and `<next-0,P1,P2,P3>` respectively, where Pi is the current BIP level for application-i. The `<pre-0,P1,P2,P3>` SDM always follows the pre-level BIP policy for the first application and the insertion policy for the remaining applications are the polices currently used in the follower sets, whereas, the `<next-0,P1, P2,P3>` SDM always follows next-level BIP for the first application and the current level BIP level for the remaining applications. Note that the hamming distance of the insertion policy between the per-application pair of SDMs is two, and each has one hamming distance with the insertion policy of the follower sets `<cur-0,P1,P2,P3>`. TAEDIP can be thought of as a cache learning scheme about each application. In this way, TAEDIP can adjust the insertion policy for different applications in the follower sets step by step and finally reach a point where the system has better performance.

4 Evaluation

In this section, we first describe the simulation environment. Then the experimental results are utilized to evaluate the effect of EDIP both for single core and quad-core systems. Finally, we analyze the hardware overhead of TAEDIP.

4.1 Simulation Setup

We use CMP$im [2], a Pin [6] based trace-driven x86 simulator, to evaluate our proposal. The simulated processor is 8-stage, 4-wide pipeline and equipped with

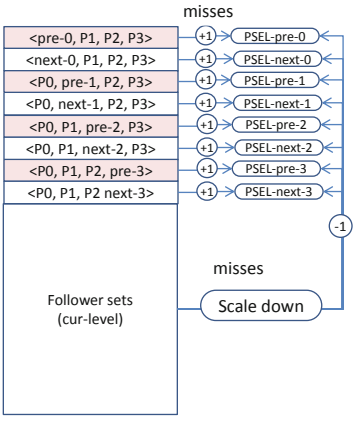


Fig. 3. TAEDIP-G

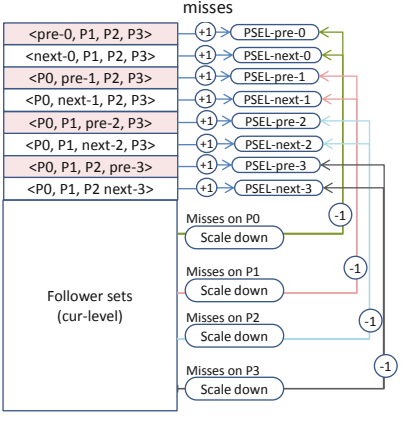


Fig. 4. TAEDIP-I

perfect branch prediction (i.e., no front-end or fetch hazards). A maximum of two loads and one store can be issued each cycle. The memory model consists of a 3-level cache hierarchy, which includes an L1 split instruction and data caches, an L2, and an L3 (Last Level) cache. All caches support 64-byte lines. The parameters of baseline system are given in Table 3.

For single-core studies we choose fifteen single-threaded applications (shown in Table 4) from SPEC2006. We first fast forward each workload for four billion cycles and then collect traces for one billion cycles. For multi-program workloads on quad-core CMP, we randomly select 10 combinations of single-threaded workloads (shown in Table 5). Simulations were run until all the single-threaded workloads reach the end of trace. If some workload reaches the end before others, the simulator rewinds its trace and restarts from beginning.

4.2 EDIP Performance for Single Core

For single core system, we simulates two EDIP schemes. The first is named EDIP-8 which use 8 BIP levels as in Table 2. The second is referred as EDIP-4 which only uses 4 odd levels in Table 2. Fig. 5 and Fig. 6 show the cache performance and the overall system performance for three policies: DIP, EDIP-8 and EDIP-4, where the baseline replacement policy is the LRU policy. EDIP gets the maximum performance with about 55.6% reduction in misses and 27.4% reduction in CPI for xalancbmk benchmark. For 9 of the 15 benchmarks, DIP, EDIP-8 and EDIP-4 have similar performance close to LRU policy. These workloads are either LRU friendly workloads or streaming workloads. For the other six workloads, EDIP gets 30.4% reduction in misses and 11.4% in CPI on average, while DIP reduces cache misses by 27.5% and CPI by 9.5%, and the results for EDIP-4 are 29.7% and 11.0% respectively. These results reflect the efficiency of refining the BIP policy.

Table 3. Baseline System Parameters

Component	Parameter
Processing core	4-wide pipeline out-of-order
Instruction latency (without miss)	1 cycle
L1 I-cache	32KB/4-way/LRU/1 cycle
L1 D-cache	32KB/8-way/LRU/1 cycle
L2 cache	256KB/8-way/LRU/10 cycles
L3 (LLC) cache (single core)	1MB/16-way /30 cycles
L3 (LLC) cache (Quad core)	4MB/16-way/30 cycles
Memory access latency	200 cycles
Cache line	64B

Table 4. Single-threaded workloads

401.bzip2	403.gcc	410.bwaves
429.mcf	433.milc	434.zeusmp
436.cactusADM	444.namd	450.soplex
456.hmmer	462.libquantum	470.lbm
471.omnetpp	482.sphinx3	483.xalancbmk

4.3 EDIP Performance for Shared Caches

We use three metrics for measuring the performance of multiple concurrently running applications: throughput, weighted speed up and fairness. The execution time is indicated by the weighted speedup metric [11], and the harmonic mean fairness metric (which is harmonic mean of normalized IPCs) balances both fairness and performance [7]. The different metrics are defined as follows:

Table 5. Multiprogrammed workloads

Mix-1	401,403,429,433	Mix-2	482,429,433,403
Mix-3	429,433,450,456	Mix-4	434,483,482,450
Mix-5	410,433,436,456	Mix-6	450,433,462,434
Mix-7	444,456,462,471	Mix-8	401,434,470,456
Mix-9	462,433,450,401	Mix-10	401,436,456,471

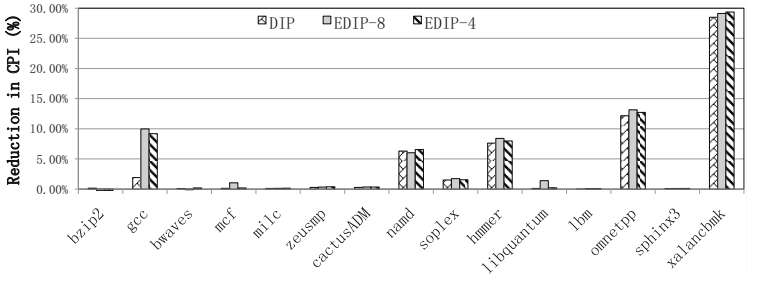


Fig. 5. Normalized CPI

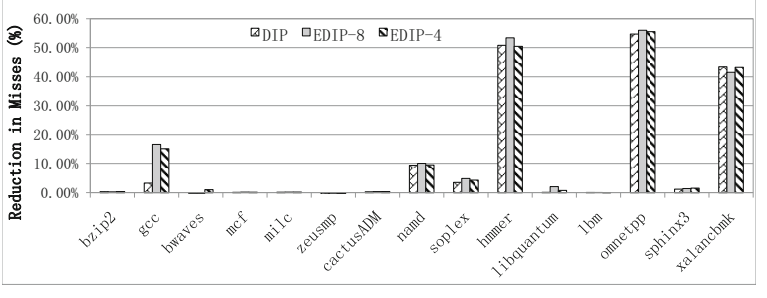


Fig. 6. Normalized L2 misses reduction

$$Throughput = \sum IPC_i \tag{1}$$

$$Weighted\ Speedup = \sum (IPC_i / SingleIPC_i) \tag{2}$$

$$HarMean = N / \sum (SingleIPC_i / IPC_i) \tag{3}$$

Fig. 7 shows the throughput of the three adaptive insertion policies (TADIP, TAEDIP-I, and TAEDIP-G) normalized to the baseline LRU policy. The x-axis represents the different workload mixes. The bar labeled geomean is the geometric mean of all 10 mixed workloads. EDIP improves throughput by more than 10% for four out of the ten workloads. For Mix-8, TAEDIP-I improves throughput by near to 20%. TAEDIP-I provides further improvement compared

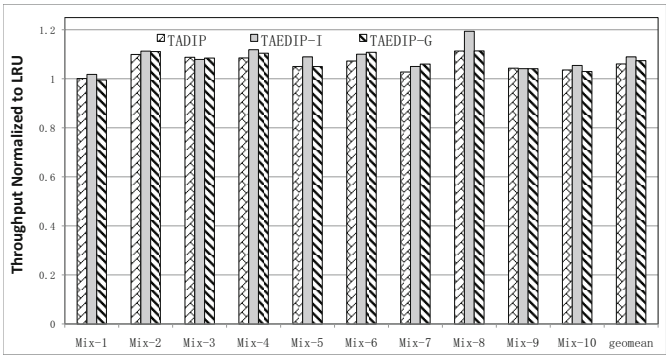


Fig. 7. Comparison of TAEDIP for Throughput Metric

to TADIP. The addition of individual feedback improves the performance of TAEDIP-I compared to TAEDIP-G except for mix3 and mix6. On average, TADIP improves throughput by 6.0%, TADIP-I improves throughput by 8.9% and TADIP-G improves throughput by 7.4%. This illustrates the importance of fine grained refining to the BIP insertion policy.

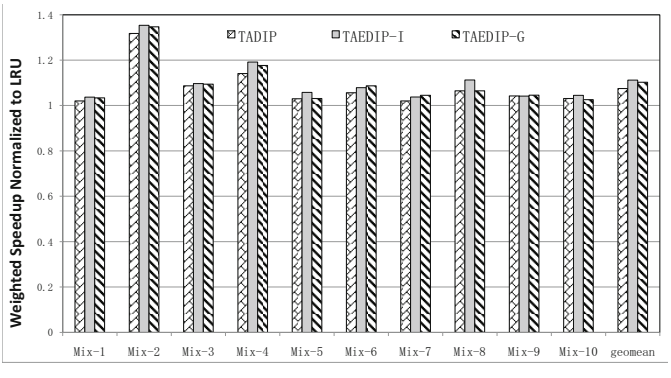


Fig. 8. Comparison of TAEDIP for Weighted Speedup

Fig. 8 shows the performance of TADIP, TAEDIP-I, and TADIP-G insertion policies on the weighted speedup metric. The values are normalized to the weighted speedup of the baseline LRU policy. Both TAEDIP-I and TAEDIP-G improves performance by more than 10% for Mix-2, Mix-4 and Mix-8. For Mix-2, TAEDIP-I improves the weighted speedup by 35%. On average, TADIP improves weighted speedup by 7.5%, TADIP-I improves weighted speedup by 11.2% and TADIP-G improves weighted speedup by 9.5%.

The fairness metric proposed by [11] considers both fairness and performance. Fig. 9 shows the performance of the baseline LRU policy, TADIP, TAEDIP-I,

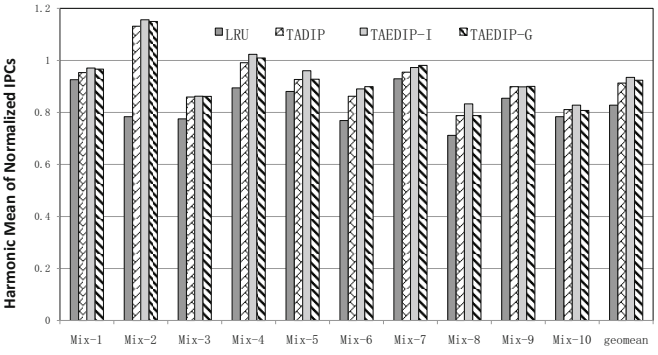


Fig. 9. Comparison of TAEDIP for Harmonic Mean Fairness

and TAEDIP-G for the fairness metric. All schemes except LRU provides more than 1 fairness for Mix-2. The baseline LRU policy has an average fairness metric value of 0.827, TADIP of 0.913, TAEDIP-I of 0.939, and TADIP-G of 0.928.

4.4 Hardware Overhead

Here we gives a summary of the overhead of different insertion policies. The LRU/LIP policy does not require any SDMs. The brute force approach [3] is impractical for large number of cores (N). TADIP need N SDMs. Both TAEDIP-I and TAEDIP-G need to record the current BIP level of each application, and require 2N SDMs, 2 for each of the N applications. TAEDIP-I needs to count the misses of each application (5-bits per thread) while TAEDIP-G only keeps the global miss (7-bits) in the follower sets. Each PSEL only takes 9-bits storage overhead. It needs additional 3-bits to record the current BIP level for each thread. The overall storage overhead is small and negligible. Note that both TAEDIP-I and TAEDIP-G incur additional logic to identify the SDMs, and the set selection logic is the same as TADIP. So the hardware overhead of EDIP for shared caches is very small, and can be used for large scale systems.

5 Related Work

There is an impressive amount of research work attempting to improve the performance of cache replacement policies. We describe the work which is most relevant to our proposal in the following paragraphs.

Lee et al. [5] proposed Least Frequently Used (LFU) replacement policy which predicts the blocks frequently accessed have a *near-immediate* re-reference interval, and the blocks infrequently accessed have a *distant* re-reference interval. LFU is suitable for workloads with frequent scans, but it doesn't work well for workloads with good recency. Then LRFU [5] is proposed to combine recency and frequency to make replacement.

Qureshi et al. [9] proposed Dynamic Insertion Policy (DIP) which places a few of the incoming lines in the LRU position instead of MRU position. DIP can preserve some of the working set in cache, which is suitable for thrash-access patterns. However, inserting lines at LRU position makes cache blocks are prone to be evicted soon and have no time to learn to retain active working set. Since DIP adopts a single insertion policy for all references of the workload, the LRU component of DIP will discard active cache blocks with single used cache lines. Jaleel et al. [3] extends DIP to make it thread-aware for shared cache multi-core.

RRIP [4] is the mostly recent proposal that predicts the incoming cache line with a re-reference interval between *near-immediate* re-reference interval and a *distant* re-reference interval. The prediction interval is updated upon re-reference. The policy is robust across streaming-access patterns and thrash-access patterns. However, it is not suitable for recency-friendly access patterns. The proposed DAI-RRP [15,16] is more robust than RRIP for it is also suitable for access patterns with high temporal-locality. In addition, the hardware overhead of DAI-RRP is comparable to RRIP, which is identical to LRU policy.

Cache partitioning is also one popular way of managing shared caches among competing applications. Cache partitioning allocates cache resources to competing applications either statically or dynamically. Stone et al. [12] investigated optimal (static) partitioning of cache resources between multiple applications when the information about change in misses for varying cache size is available for each of the competing applications. However, such information is non-trivial to obtain dynamically for all applications as it is dependant on the input set of the application. The focus of our study is to dynamically manage cache resources without requiring any prior information about the application. Dynamic partitioning of shared cache was first studied by Suh et al. [13]. They describe a mechanism to measure cache utility for each application by counting the hits to the recency position in the cache and used way partitioning to enforce partitioning decisions. The problem with way partitioning is that it requires core identifying bits with each cache entry, which requires changing the structure of the tag-store entry. Way partitioning also requires that the associativity of the cache be increased to partition the cache among a large number of applications. Qureshi et al. [8] improved on [13] by separating the cache monitoring circuits outside the cache so that the information computed by one application is not polluted by other concurrently executing applications. They provide a set-sampling based utility monitoring circuit that requires a storage overhead of 2KB per core and used way partitioning to enforce partitioning decisions. Xie et al. provided adaptive insertion policy to cover both cache capacity management and inter-core capacity stealing.

6 Conclusions and Future Work

This paper illustrates that BIP with a fixed possibility is not fit for all the memory access patterns, because it assumes a cyclic reference model where an access pattern may repeat many times. To overcome this, we refine the original BIP into

fine grained multiple BIP levels. We propose *Enhanced Dynamic Insertion Policy* (EDIP) and *Thread Aware Enhanced Dynamic Insertion Policy* (TAEDIP). By dynamically choosing an appropriate BIP level which inserts a new incoming cache block at the MRU position with specific possibilities, EDIP/TAEDIP schemes can adapt for more memory access patterns. Simulation results in single core configuration with 1MB 16-way LLC show that EDIP reduces CPI over LRU and DIP by an average of 11.4% and 1.8% respectively. On quad-core configuration with 4MB 16-way LLC. TAEDIP improves the performance on the weighted speedup metric by 11.2% over LRU and 3.7% over TADIP on average. For fairness metric, TAEDIP improves the performance by 11.2% over LRU and 2.6% over TADIP on average.

In this study, the possibility differences of two arbitrary adjacent level vary from 1/2 to 1/64, which may cause little cache performance difference for BIP-0 and BIP-1. A more flexible mechanism may be designed by carefully choosing the threshold intervals between each BIP levels. Moreover, combination of fine-grained BIP with other non-LRU policies or cache partition techniques can be our future work.

References

1. Beckmann, B.M., Marty, M.R., Wood, D.A.: ASR: Adaptive Selective Replication for CMP Caches. In: MICRO-39, pp. 443–454 (2006)
2. Jaleel, A., Cohn, R., Luk, C.K., Jacob, B.: CMP$im: A pinbased on-the-fly multi-core cache simulator. In: Workshop on Modeling Benchmarking and Simulation (MoBS) Colocated with ISCA-35 (2008)
3. Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely Jr., S., Emer, J.: Adaptive insertion policies for managing shared caches. In: PACT-17, pp. 208–219 (2008)
4. Jaleel, A.K.B., Theobald Jr., S.C.S., Emer, J.: High performance cache replacement using re-reference interval prediction (RRIP). In: ISCA-37, pp. 60–71 (2010)
5. Lee, D., Choi, J., Kim, J.H., Noh, S.H., Min, S.L., Cho, Y., Kim, C.S.: LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. IEEE Transactions on Computers 50(12), 1352–1361 (2001)
6. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI 2005, pp. 190–200 (2005)
7. Luo, K., Gummaraju, J., Franklin, M.: Balancing throughput and fairness in SMT processors. In: ISPASS, pp. 164–171 (2001)
8. Qureshi, M.K., Patt, Y.: Utility Based Cache Partitioning: A Low Overhead High-Performance Runtime Mechanism to Partition Shared Caches. In: MICRO-39 (2006)
9. Qureshi, M., Jaleel, A., Patt, Y., Steely, S., Emer, J.: Adaptive Insertion Policies for High Performance Caching. In: ISCA-34, pp. 167–178 (2007)
10. Qureshi, M.K., Lynch, D.N., Mutlu, O., Patt, Y.N.: A case for MLP-aware cache replacement. In: ISCA-33, pp. 167–178 (2006)
11. Snavely, A., Tullsen, D.M.: Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In: ASPLOS-9, pp. 234–244 (2000)
12. Stone, H.S., Turek, J., Wolf, J.L.: Optimal Partitioning of Cache Memory. IEEE Transactions on Computers 41(9), 1054–1068 (1992)

13. Suh, G.E., Rudolph, L., Devadas, S.: Dynamic Partitioning of Shared Cache Memory. Journal of Supercomputing 28(1), 7–26 (2004)
14. Xie, Y., Loh, G.H.: PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In: ISCA-36, pp. 174–183 (2009)
15. Zhang, X., Li, C., Liu, Z., Wang, H., Wang, D., Ikenaga, T.: A Novel Cache Replacement Policy via Dynamic Adaptive Insertion and Re-Reference Prediction. IEICE Transactions on Electronics E94-C(4), 468–478 (2011)
16. Zhang, X., Li, C., Wang, H., Wang, D.: A Cache Replacement Policy Using Adaptive Insertion and Re-reference Prediction. In: SBAC-PAD-22, pp. 95–102 (2010)

A Read-Write Aware Replacement Policy for Phase Change Memory*

Xi Zhang, Qian Hu, Dongsheng Wang, Chongmin Li, and Haixia Wang

Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science & Technology, Tsinghua University,
Beijing 100084, China
{zhang-xi06,hq07,lcm03}@mails.tsinghua.edu.cn,
{hx-wang,wds}@tsinghua.edu.cn

Abstract. Scaling DRAM will be increasingly difficult due to power and cost constraint. Phase Change Memory (PCM) is an emerging memory technology that can increase main memory capacity in a cost-effective and power-efficient manner. However, PCM incurs relatively long latency , high write energy, and finite endurance. To make PCM an alternative for scalable main memory, write traffic to PCM should be reduced, where memory replacement policy could play a vital role.

In this paper, we propose a Read-Write Aware policy (RWA) to reduce write traffic without performance degradation. RWA explores the asymmetry of read and write costs of PCM, and prevents dirty data lines from frequent evictions. Simulations results on an 8-core CMP show that for memory organization with and without DRAM buffer, RWA can achieve 33.1% and 14.2% reduction in write traffic to PCM respectively. In addition, an Improved RWA (I-RWA) is proposed that takes into consideration the write access pattern and can further improve memory efficiency. For organization with DRAM buffer, I-RWA provides a significant 42.8% reduction in write traffic. Furthermore, both RWA and I-RWA incurs no hardware overhead and can be easily integrated into existing hardware.

1 Introduction

Chip multiprocessor (CMP) is proposed to maintain the expected performance advances within the power and design complexity constraints. Future CMP will integrate more cores on a chip, and the increase in the number of concurrently running applications (or threads) increases the demand on storage. However, using DRAM as main memory is expensive and costs a significant portion of system power. Furthermore, the scaling of traditional memory technology is already hitting the power and cost limits [1]. It is reported that scaling DRAM beyond 40 nanometers will be increasingly difficult [2]. Hence, exploiting emerging technology is crucial to build large-scale memory system within the power and cost constraint.

* This work is supported by Nature Science Foundation of China (No. 60833004, 60970002), and the National 863 High Technology Research Program of China (No.2008AA01A201).

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 31–45, 2011.

Table 1. Comparison of Different Memory Technologies

Features	SRAM	DRAM	PCM	NAND FLASH
Density	1X	4X	<16X	16X
Read Latency *	2^5	2^9	2^{11}	2^{17}
Dyn. Power	Low	Medium	Medium read High write	Very high read Even high write
Leak. Power	High	Medium	Low	Low
Non-volatile	No	No	Yes	Yes
Scalability	Yes	Yes	Yes	Yes
Endurance	N/A	N/A	$10^8 - 10^9$	10^4
Retention	Constant power	Refresh	10yrs	10yrs

*Order of magnitude , in terms of processor cycles for a 4GHz processor.

Two promising technologies that fulfill these criteria are *Flash* and *Phase Change Memory* (PCM). NAND flash can be only used as a disk cache [3] or an alternative for disks [4] for it is not byte-addressable and is about 200X slower than DRAM. PCM, on the other hand, is only 2x-4x slower than DRAM and can provide up to 4x more density than DRAM, which makes it a promising candidate for main memories [5]. PCM has been backed by key industry manufacturers such as Intel, STMicroelectronics, Samsung, IBM and TDK as both off-chip and on-chip memory [6,7]. Table 1 lists features of several memory technologies: Static RAM (SRAM), Dynamic RAM (DRAM) and PCM based on the data obtained from literature [8,10,11].

There are several challenges to overcome before PCM can become a part of the main memory system. First, PCM being much slower than DRAM, makes a memory system comprising exclusively of PCM have much increased memory access latency. Thus, several studies proposed a properly designed main memory that is made of different memory technologies. Secondly, for its non-volatile property, PCM is unfriendly to write access. Compared to read access, write access to PCM costs too much power and latency to accomplish. In addition, the limited system lifetime due to endurance constraint is still a concern, for PCM devices can only tolerate $10^8 - 10^9$ writes per cell [9]. Therefore, the write traffic to PCM should be optimized in purpose of decreasing access latency, power and increasing lifetime.

In this paper, we focus on a memory replacement policy to decrease write traffic to PCM. Nowadays most systems use LRU or its approximations for on-chip cache replacement policy and main memory replacement policy. However, systems with PCM as main memory require a new replacement policy which considers not only hit rate but also the replacement cost. Since write access to PCM necessitates more time and energy than read access, a new replacement policy should be designed for memory level above PCM to filter out more write access requests than existing policies. Inspired by the recently proposed RRIP [19], we propose a Read-Write Aware (RWA) replacement policy, which takes into consideration the imbalance of read and write costs upon eviction. The basic idea of this method is to preserve a certain amount of dirty data lines to stay longer than clean data lines in upper level memory of PCM, thereby decreasing write backs to the lower level PCM. However, this imbalance policy may lead to

memory utilization inefficiency. More read misses may occur and total hit rate degrades. Thus the replacement policy should be delicately designed to prevent performance degradation. Through execution-driven full-system simulation, we evaluate our proposed RWA on two PCM memory architectures: one with DRAM as buffer and the other without. Results show that write traffic to PCM can be reduced by 33.1% and 14.2% respectively compared to LRU baseline, while the system performance is retained. Moreover, an Improved RWA (I-RWA) policy considering write patterns can further improve memory efficiency. I-RWA provides 11.6% reduction in write traffic, and further improves system performance in organization without DRAM buffer. For organization with DRAM buffer, I-RWA provides a significant 42.8% reduction in write traffic, without degrading performance. Furthermore, neither RWA nor I-RWA incurs hardware overhead or time overhead.

2 Background

In DRAM, each bit is stored in the form of charge in a small capacitor. DRAM requires a consistent refresh operation to sustain data and consumes a significant amount of power. Unlike DRAM, PCM is a type of non-volatile memory that can retain its data even when the power is turned off. The PCM cell consists of a standard NMOS access transistor and phase change material, GST ($Ge_2Sb_2Te_5$), as shown in Fig. 1. The phase change material is between a top and a bottom electrode with a heating element. When current is injected into the heating element, the phase change is induced. The phase change material can be switched between two states, amorphous and crystalline. The cell can be addressed using a selection transistor shown in Fig. 1(b). The read and write operations for a PCM cell are described as follows:

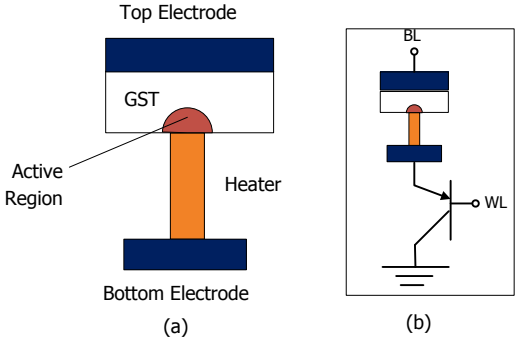


Fig. 1. PRAM Cell (a) and Transistor (b)

Write Operation: There are two kinds of PCM write operations, SET and RESET. The SET operation switches GST into crystalline phase by heating GST above its crystallization temperature. The RESET operation switches the

material into amorphous phase by melt-quenching GST. These two write operations are controlled by different electronic currents. The SET operation is controlled by moderate power and long duration of electrical pulse, and it makes the cell low-resistance. On the contrary, the RESET operation is controlled by high-power pulses which return the cell in high-resistance state.

READ Operation: Since the SET status and RESET status have a large variance on their equivalent resistance, the data stored in the cell can be retrieved by sensing the device resistance by applying very low power.

As described, compared to DRAM, the PCM write power is high, while the PCM read power is much lower. Besides, PCM consumes no refresh power, as it can retain its information permanently. In addition, the access latency of random reads/writes on PCM is slower than DRAM, and write latency is even slower than read (about an order of magnitude). In general, PCM is only 2X-4X slower than DRAM. Furthermore, PCM is denser than DRAM. With comparable size, PCM can provide up to 4X more density. That's because PCM cell can be in different degrees of partial crystallization thereby is able to store two or more bits in each cell. Finally, PCM cells can endure a maximum of $10^8 - 10^9$ writes, which poses a reliability problem. In summary, these capabilities combined with low memory cost makes PCM a promising candidate for main memory, though there are some disadvantages to overcome. In next section, we propose a method to reduce write penalty incurred by PCM.

3 Read-Write Aware Replacement Policy

In this section, we first describe overview of RWA. Then our proposed Read-Write Aware policy and its improved version are illustrated in details.

3.1 Overview

Fig. 2 shows two systems in which PCM acts as main memory. In both systems, flash is used to take place of disk as some commodity systems (e.g. MacBook

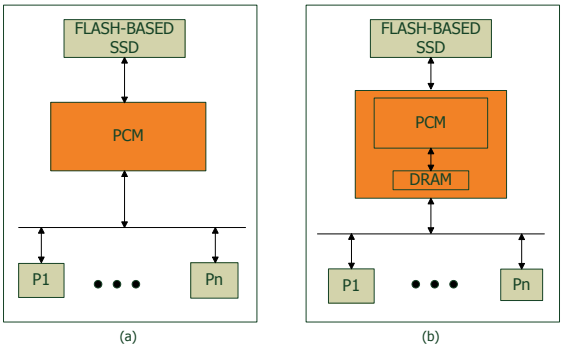


Fig. 2. (a) System with PCM (b) System with Hybrid Memory

Air laptop). PCM is used instead of DRAM to increase main memory capacity as shown in Fig. 2(a). However, the higher access latency of PCM may lead to performance degradation. Therefore, a DRAM buffer can be introduced into system as the cache of PCM (shown in Fig. 2(b)). In such a hybrid main memory, system can get benefits from both the large capacity of PCM and the low access latency of DRAM. It is shown that a relative small DRAM buffer (3% size of the PCM storage) can filter most of the accesses to PCM [10]. This organization is similar to the organization proposed in [10], but is different in access granularity and data placing sequence. In the hybrid system, PCM acts main memory as DRAM in a traditional system. Therefore, it is managed by Operating System (OS). In contrast, the DRAM cache is not visible to OS and organized by DRAM controller (In our system DRAM is off-chip, however, on-chip embedded DRAM is also feasible as the organization in IBM POWER7 [12].). DRAM is organized at cache block granularity as same as SRAM cache on chip. The reason why we don't use page granularity is that when replacement occurs in DRAM, only dirty lines are written back to PCM instead of dirty pages. Thus, write traffic to PCM is reduced. This mechanism requires Dirty Bits attached to each DRAM cache line. In addition, Valid Bit and Least Recently Used (LRU) Bits are also required per cache line. In order to reduce access latency, the tags of DRAM buffer can be made of SRAM.

We assume write-back SRAM cache and DRAM cache considering the endurance limit of PCM. PCM is written only when dirty data lines are replaced from the upper level memory due to capacity conflicts. The baseline replacement policy of on-chip SRAM cache and off-chip DRAM cache is LRU, which doesn't consider the imbalance of replacement cost for read and write request. In order to reduce write-back traffic to PCM, a novel Read-Write Aware replacement policy is proposed for the memory level above PCM. The goal of our proposal is to reduce write traffic from this upper level memory to PCM, without performance degradation. In organization shown in Fig. 2(a), the upper level memory of PCM is the on-chip last-level cache, whose replacement policy decides the amount of write traffic to PCM. To better illustrate our policy, we assume there are two cache levels on chip, and the last-level cache is L2 cache. Nonetheless, RWA can be applied in other memory organizations. In general, L2 cache receives three types of request from L1 cache: load, store and write back. L2 cache writes to PCM when dirty data lines in it are replaced due to conflicts. Our RWA policy is adopted in L2 cache and distinguishes read and write requests arriving at L2 cache. RWA in L2 cache impacts the amount of write traffic to PCM by deciding which and when dirty data lines are replaced.

Please note that, in *Read-Write Aware* policy, *Read* refers to load or store request, while *Write* refers to write back request due to replacement. In other words, load and store requests are both treated as read operation in our policy, for they don't modify data lines. Only write back request actually produces dirty data lines, and the data lines will be replaced to the next level PCM in future. Therefore, we just distinguish read operations (load/store) from write operations (write back) in our policy. In addition, for organization with DRAM buffer shown in

Fig. 2(b), RWA can be used both in on-chip last level cache and DRAM buffer. Moreover, RWA is adaptive for both inclusive and non-inclusive memory.

3.2 Read-Write Aware Replacement Policy

To reduce the write traffic to PCM, we propose RWA policy for memory level above PCM. The goal of our proposal is to reduce evictions of dirty data lines from the upper level memory of PCM. It is accomplished by retaining dirty cache lines long enough to satisfy subsequent write backs.

The recently proposed cache replacement policy using Re-Reference Interval Prediction (RRIP) [19] aims at preventing blocks with a distant re-reference interval from polluting the cache. An M-bit register per cache block is used to store its Re-reference Prediction Value (RRPV), which predicts the re-reference interval is *near-immediate*, *long* or in the *distant* future. Blocks with *distant* RRPV will be evicted first when confliction, then will be the blocks with *long* RRPV. Finally *near-immediate* RRPV blocks will be evicted. By always inserting new blocks with a *long* RRPV, new blocks are earlier to be evicted compared to LRU. Thus, RRIP prevents cache blocks with a *distant* re-reference interval from evicting blocks having a *near-immediate* re-reference interval, which is fit for workloads with stream access and mixed access patterns.

Inspired by RRIP, we proposed RWA, which distinguishes read and write request by setting different RRPVs. We assume an n-way associativity cache. Similar to RRIP, RRPV 0 means the data line is predicted to be re-referenced in *near-immediate* term and has the lowest priority to be evicted, while RRPV $(n-1)$ means the data line is predicted to be re-referenced in *distant* future and has the highest priority to be evicted. RRPV between 0 and $(n-1)$ means the data line has medium re-reference interval. The lager RRPV is, the higher priority the data line will be evicted. To illustrate RWA policy clearly, it is decomposed into three parts: *the insertion policy*, *the promotion policy* and *the victim selection policy*, which are described as follows (assuming n-way cache):

The Insertion Policy. On a load or store miss, RRPV of the new block is set to $(n-2)$. On a write back miss, RRPV of the new block is set to 0.

The Promotion Policy. On hit, no matter it is a load hit, store hit or write back hit, RRPV of the hit line is set to 0.

The Victim Selection Policy. The data line whose RRPV is $(n-1)$ is victim. If there are no data lines whose RRPV is $(n-1)$, all the RRPVs in this set are incremented until some RRPV reaches $(n-1)$. If there are multiple data lines whose RRPVs are $(n-1)$, we just select one randomly.

Behavior of RWA policy with 8-way DRAM (or last-level cache) is shown in Fig. 3.

3.3 Improved Read-Write Aware Replacement Policy

In the proposed RWA policy, all dirty data lines are protected to decrease replacements, which may impact the system performance. For example, newly

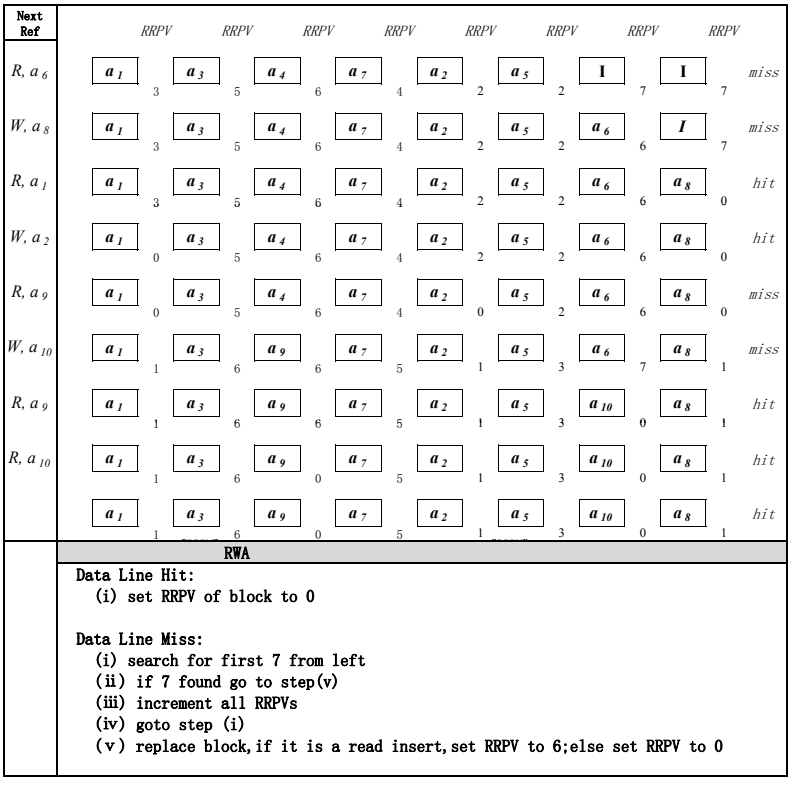


Fig. 3. Example of Read-Write Aware policy

installed dirty data line is given the lowest priority to be evicted when write back miss occurs. If there are subsequent write backs to this data line in short term, some replacements can be avoided. However, if it is a single-use write back (i.e. no subsequent write backs to this data line), according to the promotion policy, it won't be replaced until its RRPV is incremented to $(n-1)$ after a lot of accesses. As a result, clean blocks and hot dirty lines with multiple hits are prone to be replaced, which leads to memory under-utilization and thus impacts system performance.

To prevent single-use dirty lines from wasting capacity, an Improved RWA (I-RWA) policy is proposed. In such mechanism, the insertion policy is different from that of RWA. Newly dirty lines will not be given lowest priority as in RWA. Instead, it is given a medium RRPV value. This may prevent single-use dirty lines from polluting cache, and meanwhile gives the new line some time to receive subsequent hits. Once it is re-referenced, its RRPV is promoted to 0 as in RWA. As a result, only dirty lines with multiple uses are protected, which increase memory efficiency.

However, always giving new dirty lines medium RRPV will shrink the total capacity of dirty lines. Because with the new insertion policy, write accesses

have similar operations as read access, and dirty lines are not more protected than clean lines now. Consequently, more dirty data line replacements may be triggered. To address this limitation, load/store hit promotion policy is improved. When load/store hit, RRPV of the hit data line is set to some medium value instead of 0. While the promotion policy for write hit remains the same as RWA. In summary, I-RWA is described as follows (assuming n-way cache):

The Insertion Policy. On a load or store miss, RRPV of the new block is set to $(n-2)$. On a write back miss, RRPV of the new block is set to $(n-3)$.

The Promotion Policy. On read (load/store) hit, RRPV of the hit line is set to 3; on write hit, RRPV of the hit line is set to 0.

The Victim Selection Policy. The same as RWA policy.

With the I-RWA policy, single-use and multiple-use dirty lines are distinguished and only multiple-use dirty lines are more protected than other data lines, that increases memory efficiency.

Assuming n-way cache, a $log_2 n$ bits register is required for each data line to store RRPV. Thus, our proposed RWA (I-RWA) policy has no hardware overhead compared to LRU, and can be applied to memory levels above PCM, no matter it is on chip or off-chip, DRAM or SRAM. In organization shown in Fig. 2(a), RWA (I-RWA) can be used in last-level cache on-chip. In organization shown in Fig. 2(b), RWA (I-RWA) can be used in both the last-level cache and DRAM cache. In next section, RWA and I-RWA will be evaluated on both of the two memory organizations.

4 Evaluation

4.1 Experiment Setup

By using full-system simulation based on Simics [13] and the GEMS toolset [14], we evaluate our proposed RWA policy and I-RWA policy on both the memory architectures shown in Fig. 2.

The parameters of baseline configurations are given in Table 2. Note that we have two baseline systems, one with DRAM cache and one without. Except

Table 2. Base line System Parameters

Component	Parameter
Processing Core	8-core/Sparc V9 ISA/in-order/
L1 Cache	I-cache and D-cache each 16KB, 2-way, 64B linesize, writeback policy, LRU, 2 cycles
L2 cache	4M, 8 banks, 8-way, 64B linesize, writeback policy, LRU, 15 cycles
DRAM cache	256MB, 8 banks, 8-way, 64B linesize writeback policy, 200 cycles
PCM memory	8GB, 1024 cycles read latency, 4096 cycles write latency, 4 KB page size
Flash-based SSD	100K cyckes, 100% hit rate
Network Configuration	4x2 Mesh, 1-cycle hop latency

DRAM, all parameters of the two baseline systems are the same. It is an eight-core CMP system with simple in-order core modal. The L1 cache is private, while L2 cache, DRAM cache and PCM memory are all shared by the eight-core. LRU policy is adopted in both L2 cache and DRAM cache, while PCM memory uses NRU [27] page replacement policy. Detailed memory controllers are also simulated. Since PCM is emerging memory technology, the projection of its feature tends to be more varied, however, the parameters are chosen in-line with other researchers' assumptions.

We study our design using ten scientific workloads from SPLASH-2 [25] and two workloads from PARSEC [26] as shown in Table 3.

Table 3. Workloads description

Workload	Problem Size
FFT	256K points
LU	512*512 matrix, 16x16 blocks
RADIX	1048576 keys, radix=1024
RAYTRACE	Teapot.env
OCEAN	514 x 514 ocean
WATER	512 molecules
BARNES	16K particles
CHOLESKY	tk15.o, Postpast partition size:32
FMM	16K particles
VOLREND	head_scaleddown2, 4x4 image block size
STREAMCLUSTER	simsmall
FLUIDANIMATE	simsmall

4.2 Results on the Organization without DRAM Buffer

In this section, we evaluate performance of RWA and I-RWA implemented on L2 cache, based on the tradition organization without DRAM cache as shown in Fig. 2(a). In baseline system, the L2 cache replacement policy is LRU. Fig. 4 presents the write traffic to PCM normalized to the baseline system. Nine of ten workloads can get benefits from RWA and I-RWA, while only in *cholesky* write traffic is slightly increased by less than 5%. Overall, the write traffic is reduced by 14.2% with RWA policy and by 11.6% with I-RWA. I-RWA reduction is less than that of RWA, because dirty lines in RWA are more protected than in I-RWA. However, compared to RWA, I-RWA is able to prevent single-use dirty lines from wasting memory capacity, and thereby increases memory efficiency. Translating this efficiency to system performance, I-RWA achieves 2.1% performance improvement over RWA, as is shown in Fig. 5.

Fig. 5 shows execution time normalized to baseline system. On average, RWA has almost the same execution time as baseline, with only a negligible difference of 0.6%. I-RWA can reduce execution time by 1.5%. For a specific workload, RWA may outperform I-RWA when there aren't many single-use dirty lines, e.g. *clolesky*. Therefore, RWA and I-RWA are efficient for different write access patterns. Since our policy is used in last-level cache (and also in DRAM cache in next section), after being filtered by upper memory levels, there is a great probability of single-use lines existing. As a result, I-RWA receives more benefits on system performance over RWA.

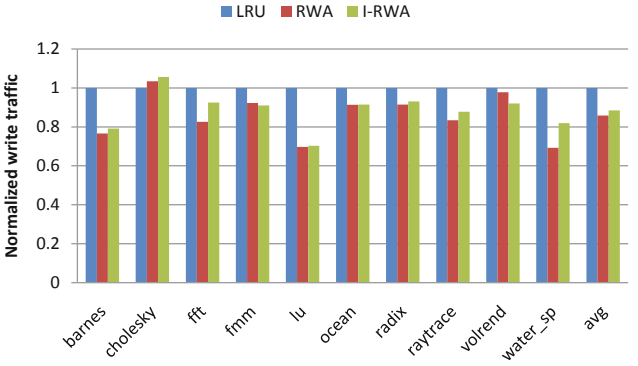


Fig. 4. Normalized write traffic to PCM without DRAM buffer

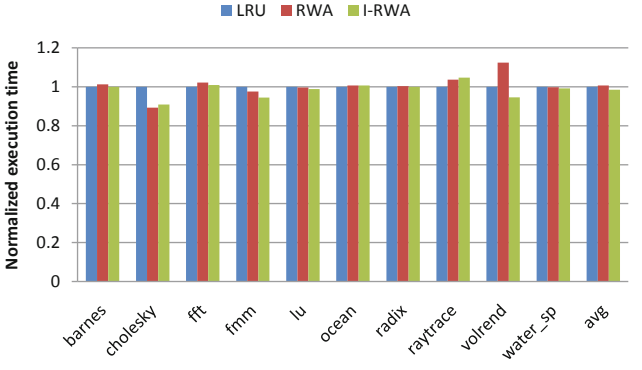


Fig. 5. Normalized Execution Time without DRAM buffer

4.3 Results on the Organization with DRAM Buffer

Since using DRAM as buffer can filter a significant amount of requests to PCM, workloads with small working set which have few requests to PCM are not evaluated here. As a result, only five workloads from SPLASH-2 are reserved. To better demonstrate the results of our proposal, we randomly select two workloads (*fluidanimate* and *streamcluster*) with large working set from PARSEC benchmark as a supplementary. The reason why we don't choose more workloads is the time constraint. Simulating a single workload takes tens of hours to accomplish.

In such hybrid organization, both the L2 cache and DRAM adopt RWA (or I-RWA) policy. Fig. 6 presents the normalized write traffic to PCM. Overall, RWA can reduce write traffic by 33.1%, while the reduction with I-RWA is 42.8%. The significant reduction shows that using our proposal in DRAM cache

together with L2 cache can further decrease write backs to PCM. Furthermore, I-RWA outperforms RWA by 9.7% on average. The reason is that, when write back requests arrive at DRAM buffer, it has been filtered by two upper levels (L1 cache and L2 cache). As a result, there is a significant amount of single-use dirty lines, and thereby I-RWA receives more benefits due to its insertion component. Single-use dirty data lines are evicted quickly, and consequently multiple-use dirty data lines are reserved longer to receive more subsequent hits. As a result, dirty lines evictions are reduced. In summary, I-RWA is more suitable for DRAM cache than RWA.

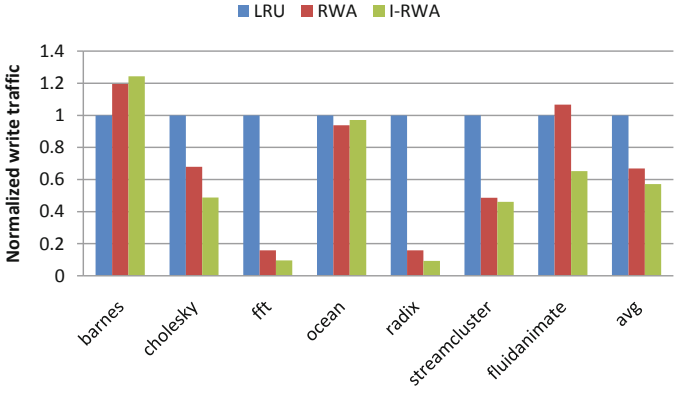


Fig. 6. Normalized write traffic to PCM with DRAM buffer

Among all the workloads, only in *barnes* both RWA and I-RWA increase write traffic. However, Fig. 5 shows that barnes is able to get benefit with our policy in L2 cache. Thus, it implies that the extra evictions are incurred by our policy equipped in DRAM cache. The possible reason is that, when running *barnes*, most of dirty lines in DRAM receive only several re-references. Thus, after it receives the last reference (and then dead), it is not easy to be evicted due to insertion policy of RWA (I-RWA). That's because a lot of new lines are inserted with higher priority to be evicted than re-referenced data lines . On the contrary, with LRU, dead dirty lines are easier to be evicted, which improves the memory efficiency. Nonetheless, RWA (or I-RWA) is able to receive significant benefits on average and from all other workloads.

Fig. 7 shows the normalized execution time compared to baseline. On average, RWA can reduce execution time by 1.3% while I-RWA provides 1.0% reduction. Neither of them increase execution time. For different workloads, RWA and I-RWA have different effects according to different write patterns. Dirty lines with many re-references are more suitable for RWA, while dirty lines with seldom re-references are more suitable for I-RWA.

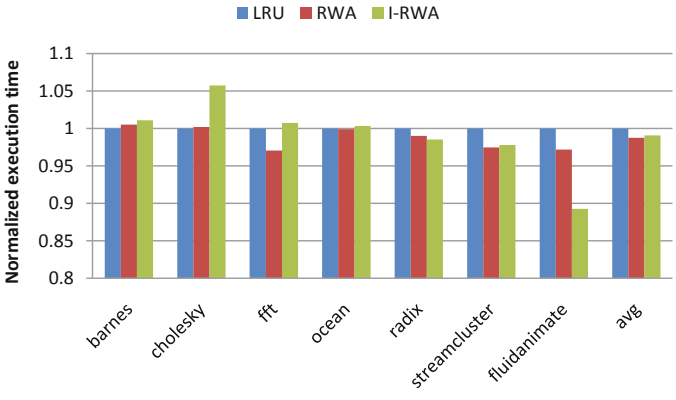


Fig. 7. Normalized Execution Time with DRAM buffer

5 Related Work

Several previous research focused on reducing high latency incurred by PCM. Related work [10,20] have proposed hybrid architecture together with buffers and intelligent scheduling to bridge the latency gap. These techniques focused on write access, for it is much slower than read access and not in the critical latency path. Qureshi et al. [10] adopted a smaller DRAM acting as a cache for a large PCM main memory. Qureshi et al. [24] proposed adaptive Write Cancelation policies combined with Write Pausing policies to improve read performance. Wu [23] et al. proposed read-write aware Hybrid Cache Architecture, where a single level cache can be partitioned into read region (SRAM) and write region (PCM or MRAM). Then cache line migration policy is proposed to move data between regions. However the policy is only fit for its specific organization and requires extra hardware overhead and instruction extension. Our proposal can be applied to any organizations without hardware overhead or instruction extension. Because PCM has significantly less write endurance compared to DRAM, reducing write traffic to PCM is of great importance. Qureshi et al. [10] proposed lazy write organization, which doesn't write back unmodified page in DRAM. Flip-N-Write [22] examines new word with original word to reduce the number of bits written to.

There is also an impressive amount of research work attempting to improve the performance of cache replacement policies, and we only describe the work which is most relevant to our proposal. Qureshi et al. [18] proposed Dynamic Insertion Policy (DIP) which places a few of the incoming lines in the LRU position instead of MRU position. DIP is suitable for thrash-access patterns. RRIP [19] is the mostly recent proposal that predicts the incoming cache line with a re-reference interval between near-immediate re-reference interval and a distant re-reference interval. The prediction interval is updated upon re-reference. The policy is robust across streaming-access patterns and thrash-access patterns. Both of the

methods are used for removing the dead blocks and increase hit rate. However, our proposal considers different miss costs of read and write request caused by PCM, and delicately select victims between dirty lines and clean liens.

There are also some flash-aware replacement polices which give clean pages high priority to evict. In CFLRU [15], there is a clean-first region at the LRU end of LRU list, where clean pages are always selected as victims over dirty pages. However, single-use pages are also preserved, and thus buffer capacity is wasted. LRU-WSR[16] is based on LRU and Second Chance. It considers using frequency and evicts clean and cold-dirty pages and keeping hot-dirty pages in buffer as long as possible. However, it doesn't address the problem of write patterns and requires extra hardware overhead. REF [17] has a victim window similar to CFLRU. However, it doesn't distinguish clean and dirty states of pages. Our proposal considers both the write patterns and asymmetry of clean and dirty states, without extra overhead. In addition, all the above three policies are optimized for flash memory in page or coarser granularity. Our proposed policy aims at PCM memory, and can be used both on chip and off-chip at any data granularity.

6 Conclusions and Future Work

We have presented Read-Write Aware policy (RWA), a last-level cache and DRAM cache replacement policy that reduces write traffic to PCM without hurting performance. Our execution-driven full-system simulation results on an 8-core CMP show that for memory organizations with and without DRAM buffer, RWA can achieve 33.1% and 14.2% reduction in write traffic to PCM respectively, while keeping the system performance not impacted. In order to increase memory utilization efficiency, I-RWA is proposed which prevents single-use dirty lines from wasting capacity. In organization without DRAM buffer, I-RWA provides around 11.6% reduction in write traffic, and further improves system performance. For organization with DRAM buffer, I-RWA provides a significant 42.8% reduction in write traffic, without degrading performance. Furthermore, neither RWA nor I-RWA incurs hardware overhead or time overhead. Therefore, the proposals are practical policies and can be easily integrated into existing hardware.

Given space and time, we could have evaluated RWA with more kinds of workloads , and shown more thorough sensitive study of RWA, e.g., the impact of power reduction, the impact of insertion positions and promotion positions, the impact of prefetch, the impact of write queue policy. In addition, since RWA and I-RWA are suitable for different access patterns, designing a dynamic policy which intelligently selects the policy best suited to the workload could be our future work. Overall, we believe this research is vital to improve future PCM performance.

References

1. Lefurgy, C., et al.: Energy management for commercial servers. IEEE Computer 36(12), 39–48 (2003)
2. Int'l Technology Roadmap for Semiconductors:Process Integration, Devices, and Structures, Semiconductor Industry Assoc. (2007), http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_PIDS.pdf
3. Kgil, T., Roberts, D., Mudge, T.: Improving NAND Flash Based Disk Caches. In: The 35th International Symposium on Computer Architecture, pp. 327–338 (2008)
4. Wu, M., Zwaenepoel, W.: eNVy: A Nonvolatile, Main Memory Storage System. In: ASPLOS-VI, pp. 86–97 (1994)
5. Raoux, S., et al.: Phase-change random access memory: A scalable technology. IBM Journal of R. and D. 52(4/5), 465–479 (2008)
6. Kanellos, M.: IBM Changes Directions in Magnetic Memory (August 2007), http://news.cnet.com/IBM-changes-directions-in-magneticmemory/2100-10043-6203198.html
7. Intel. Intel, STMicroelectronics Deliver Industry's First Phase Change Memory Prototypes. Intel News Release (February 6, 2008)
8. Wu, X., et al.: Hybrid cache architecture with disparate memory technologies. In: ISCA 2009, pp. 34–45 (2009)
9. Kang, D.H., et al.: Two-bit Cell Operation in Diode-Switch Phase Change Memory Cells with 90nm Technology. In: IEEE Symposium on VLSI Technology Digest of Technical Papers, pp. 98–99 (2008)
10. Qureshi, M., et al.: Scalable high performance main memory system using phase-change memory technology. In: ISCA-36 (2009)
11. The Basics of Phase Change Memory Technology, http://www.numonyx.com/Documents/WhitePapers/PCM_Basics_WP.pdf
12. Kalla, R., Sinharoy, B., Starke, W.J., Floyd, M.: Power7: IBM's Next-Generation Server Processor. IEEE Micro. 30(2), 7–15 (2010)
13. Virtutech AB. Simics Full System Simulator, http://www.simics.com/
14. Wisconsin Multifacet GEMS Simulator, http://www.cs.wisc.edu/gems/
15. Park, S.-Y., Jung, D., Kang, J.-U., Kim, J.-S., Lee, J.: CFLRU: a replacement algorithm for flash memory. In: Proc. of International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp. 234–241 (2006)
16. Jung, H., et al.: LRU-WSR: integration of LRU and writes sequence reordering for flash memory. Trans. on Cons. Electr. 54(3), 1215–1223 (2008)
17. Seo, D., Shin, D.: Recently-evicted-first buffer replacement policy for flash storage devices. Trans. on Cons. Electr. 54(3), 1228–1235 (2008)
18. Qureshi, M., Jaleel, A., Patt, Y., Steely, S., Emer, J.: Adaptive Insertion Policies for High Performance Caching. In: ISCA-34, pp. 167–178 (2007)
19. Jaleel, A.K.B., Theobald Jr., S.C.S., Emer, J.: High performance cache replacement using re-reference interval prediction (RRIP). In: ISCA-37, pp. 60–71 (2010)
20. Dhiman, G., et al.: PDRAM: A hybrid PRAM and DRAM main memory system. In: DAC 2009, pp. 664–669 (2009)
21. Zhou, P., et al.: A durable and energy efficient main memory using phase change memory technology. In: ISCA-36 (2009)
22. Cho, S., et al.: Flip-N-Write: A simple deterministic technique to improve pram write performance, energy and endurance. In: MICRO-42 (2009)

23. Wu, X., et al.: Power and Performance of Read-Write Aware Hybrid Caches with Non-volatile Memories. In: DATE 2009 (2009)
24. Qureshi, M.K., Franceschini, M., Lastras, L.: Improving Read Performance of Phase Change Memories via Write Cancellation and Write Pausing. In: HPCA-16 (2010)
25. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. SIGARCH Comput. Archit. News 23(2), 24–36 (1995)
26. Bienia, C., Kumar, S., Clara, S., Singh, J.P., Li, K.: The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: PACT-17, pp. 272–281 (2008)
27. UltraSPARC T2 supplement to the UltraSPARC architecture 2007. Draft D1.4.3 (2007)

Evaluating the Performance and Scalability of MapReduce Applications on X10*

Chao Zhang, Chenning Xie, Zhiwei Xiao, and Haibo Chen

Parallel Processing Institute, Fudan University

Abstract. MapReduce has been shown to be a simple and efficient way to harness the massive resources of clusters. Recently, researchers propose using partitioned global address space (PGAS) based language and runtime to ease the programming of large-scale clusters. In this paper, we present an empirical study on the effectiveness of running MapReduce applications on a typical PGAS language runtime called X10. By tuning the performance of two applications on X10 platforms, we successfully eliminate several performance bottlenecks related to I/O processing. We also identify several remaining problems and propose several approaches to remedying them. Our final performance evaluation on a small-scale multicore cluster shows that the MapReduce applications written with X10 notably outperform those in Hadoop in most cases. Detailed analysis reveals that the major performance advantages come from a simplified task management and data storage scheme.

1 Introduction

The continuity of Moore's law in the multicore era provides an increasing number of resources to harness. However, this also creates grand challenges to programmers, as the increasing number of CPU cores requires programmers to understand and write parallel programs. However, previous evidences indicate that parallel programming is notoriously hard. Hence, an ideal scheme would be providing a high-level programming model for average programmers and hiding them from complex issues such as managing parallelism and data distribution using a runtime library.

MapReduce [1], proposed by Google, is a model to program large-scale parallel and distributed systems. Mostly, programmers only need to abstract an application into a Map and a Reduce phases, while letting the underlying runtime manage parallelism and data distribution. Due to its elegance and simplicity, MapReduce has been shown as a simple and efficient way to program many applications.

To provide programmers with more fine-grained control over parallelism and data distribution, yet still maintaining good programmability, *partitioned global address space (PGAS)* [2] based languages and runtime have long been the research focus

* This work was funded by IBM X10 Innovation Faculty Award, China National Natural Science Foundation under grant numbered 61003002, a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100, a research grant from Intel, Fundamental Research Funds for the Central Universities in China and Shanghai Leading Academic Discipline Project (Project Number: B114).

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 46–57, 2011.

and have been developed in various languages such as Unified Parallel C, Fortress and Chapel. X10 [3] is one in the PGAS families that aims at providing high productivity and scalability for parallel programming. It supports a variety of features such as place, activity, clock and finish. X10 has also provided its own constructs to write data-parallel applications such as MapReduce. However, currently, there is little study on the performance and scalability of MapReduce applications written with X10 on clusters.

In this paper, we evaluate two MapReduce applications, WordCount and DistributedSort, on X10 platforms. By analyzing the performance of the two applications, we eliminate several performance bottlenecks related to I/O processing. During our study, we also identify some remaining problems and propose several approaches to remedying them.

We also compare the two applications written with X10 with those in Hadoop. In our evaluation environment, we use a small-scale 7-node cluster, with each node 24 cores and 64 GB memory. The X10 MapReduce applications and the Hadoop benchmarks are both evaluated on this cluster. Our evaluation results show that the two MapReduce applications with X10 notably outperform those in Hadoop. Detailed analysis reveals that the major performance advantages come from simplified task management and data storage schemes.

The rest of the paper is organized as follows. Section 2 presents the background information about X10's main features, its basic grammar and an overview of Hadoop. In section 3, we use X10 to implement two MapReduce applications, and the section 4 discusses the optimization of the two applications. A detailed performance evaluation of the two applications is presented in section 5, as well as a comparison with those of Hadoop's applications. In section 6, we present several optimization suggestions to remedy the remaining problems uncovered based on our study for improvement of performance and scalability of X10 applications. Section 7 discusses the related work and finally we conclude the paper in section 8.

2 Background

This section mainly introduces the fundamentals of X10 languages, as well as its runtime and collecting-finish framework, and the simple overview of the implementation of Hadoop.

2.1 An Overview of X10

X10 is a new language and runtime to program Non-Uniform Cluster Computing (NUCC) [3] systems, with the goal of both good programmability and performance scalability. It supports the *asynchronous partitioned global address space (APGAS)* programming model [2], which provides a global shared address space but partitions the address space into different portions according to system topology. The data is distributed into the partitioned address spaces called Places. Remote accesses to data in a different place are through one-directional communication primitives. To execute a series of instructions, a program declares many activities, which can be either synchronous or asynchronous. The activities can also be spawned on remote places. The

basic structure of X10 is shown in Figure 1. The number of places remains the same during the lifetime of a program, which means no place is dynamically added once a program begins to run. To schedule numbers of activities, the X10 runtime provides a work-stealing algorithm that balances the execution of activities on different workers. These allow control of parallelism, data accesses of non-uniformed memory, reducing of the cost for remote accesses and optimal task scheduling.

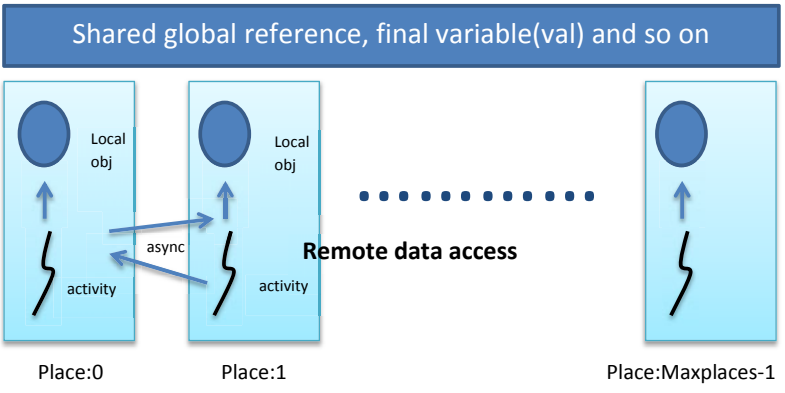


Fig. 1. Basic structure of X10 places, activities and remote data accesses

Many programming models can be subsumed under the APGAS model. For example, the MapReduce programming model can be implemented using the collecting-finish framework in X10. The followings list some important abstraction in X10 [3]:

* An ***at(p) S*** statement assigns a specific place p to execute statement S. The current activity is blocked until S terminates.
* An ***async S*** statement launches an activity and executes statement S without blocking the current activity.
* A ***finish S*** statement is used to block the current activity and wait for all activities in S to terminate.
* ***Collecting-finish*** provides an interface of Reducible[T], of which T can be any reduce unit that is to be merged by the reducer according to the method operator *this()* implemented by user (an example is shown in Figure 2). The method accepts two reduce unit and returns the reduced output in the same format. User also needs to implement an *zero()* method to define the default empty result. Meanwhile, user should also program in the finish (reducer) {} way, which may launch a number of activities and each activity can offer the intermediate value in the format of the reduce unit. The collecting-finish scheduler collects the data from the body of the finish (reducer) {} on each place. This process resembles the Reduce option in MapReduce.

```
val r = new Reducer();
      val result = finish(r){
      //do the data collecting process
      ……
      offer(partResult:T);
}
```

```
class def Reducer extends Reducible[T]{
  def zero(){
    return an instance of T that is empty.
  }
  def operator this(x:T,y:T){
    …… // do the merge option
  return (z:T);
  }
}
```

Fig. 2. An abstraction frame of collecting-finish

2.2 Hadoop

Hadoop [4] is an open-source implementation of MapReduce on clusters. A typical Hadoop deployment consists of a master and multiple slaves. The master schedules MapReduce jobs, while slaves perform the actual computation. Hadoop stores both input and output files on Hadoop Distributed File System (HDFS). HDFS stores files as blocks and replicates them to multiple nodes. Hadoop could gain block locations from HDFS and thus assign tasks to nodes storing the input data. A MapReduce job is divided into a number of Map and Reduce tasks in Hadoop. The Map task loads data from HDFS, performs the map function and flushes the intermediate data onto the local disk. The Reduce task fetches its portion of the results of all other Map tasks, performs the reduce function and saves the final results back to HDFS.

3 Implementation of Two MapReduce Applications Using X10

In this section, we discuss the design and implementation of the two X10 applications using the MapReduce programming model.

3.1 Execution Overview

Typical MapReduce applications consist of the following phases: split, map, reduce and combine or merge. The two main processing parts are the map and reduce functions. The map function emits a set of $\langle key, value \rangle$ pairs and the reduce function processes all the pairs to get the sum of values according to the key.

The two X10 MapReduce applications are written in this programming style. Figure 3 shows the overall workflow of the applications. During the program's lifetime, it goes through the following steps:

1. The splitter splits the input data into N pieces according to the number of places. The size of each piece of data is evenly divided and may be dynamically adjusted.
2. Each split of the data is assigned to a place, in which there are some pre-allocated workers. The input data is partitioned again according to the number of workers as done in step 1.
3. Once a worker gets the input data from the splitter, it scans the content and begins the map work. It produces a set of $\langle key, value \rangle$ pairs and passes them to the reducer.

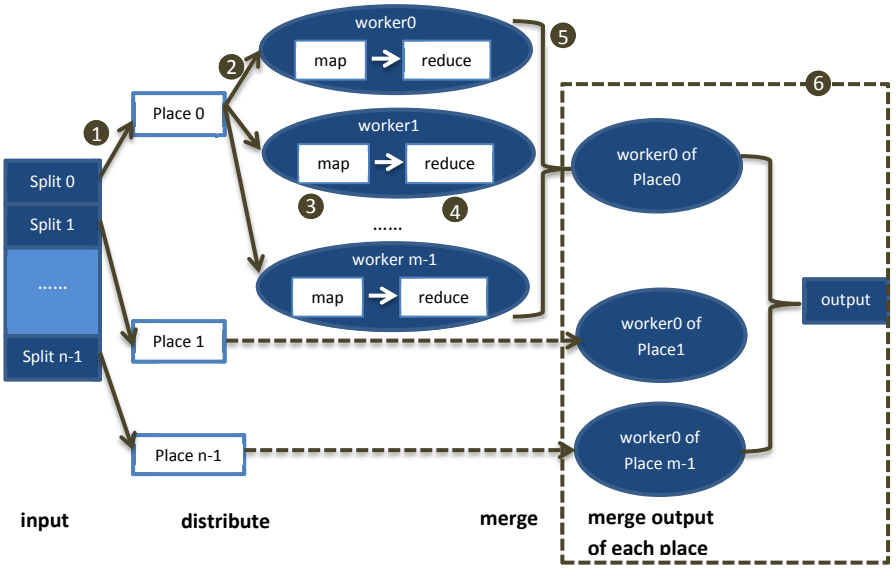


Fig. 3. Execution overflow of the two X10 MapReduce applications

4. The reduce worker gets the input pairs and aggregates the value according to the keys. After the entire map and reduce tasks complete, there are M*N pieces of output data, where M is the number of works and N is the number of places.
5. The merge work is divided into two steps. Firstly, in each place, there are M pieces of output data from the reduce worker. M/2 workers are spawned to do the merge work in parallel. At the end of this step, the intermediate value containing all the data processed by this place is ready.
6. This step collects data from all N places. After all the steps successfully completed, we can get the final result of a list of $\langle key, values \rangle$ pairs in place 0.

3.2 Data Structures

For simplicity, we use arrays to store the data. Every map and reduce worker holds an array of the $\langle key, value \rangle$ pairs. Thus, the computation can be done in parallel without synchronization since the arrays are independent. The keys are kept in the alphabetical order for simple management and binary search is used to scan arrays.

3.3 Implementing WordCount and DistributedSort

WordCount: WordCount counts the number of occurrences for each word in files. First, Place0 splits the input data into N pieces according to the number of places as shown in Figure 4. Then, it sends the split information containing the start position and the length of the data to each place. Once a place gets the information, it reads the file from

```
1   //define the reducer for collecting-finish
2   r = new Reducer();
3   result = finish(r){
4   for(p in Place.places()){
5     async at(p){
6       for(t in threads){
7         intermediate =
8            map(data_of_this_thread);
9         offer(intermediate);
10      }
11    }
12  }
```

```
1   def map(arg){
2     read(); // split data into set of <word,1>
3     reduce();//combine value of same word
4   }
5
6   def class Reducer extends Reducible[T]{
7     def apply(x:T,y:T){
8       merge_offered_intermediate(x,y);
9     }
10  }
```

Fig. 4. Splitter, map, reduce and collecting-finish of WordCount

local disks and spawns some workers to do the map and reduce operations (Figure 4(a), Line 7). After the completion of map and reduce phases, merge workers are invoked to produce the intermediate results (Figure 4(a), Line 9). Finally, the place uses the collecting-finish mode to offer the result to the master (Figure 4(a), Line 9) and the master schedules some places to do the reduce work and gets the final output.

DistributedSort: DistributedSort sorts a set of 100-byte records of which 10 bytes are used as keys. Different from WordCount, the keys of DistributedSort are all unique, which means DistributedSort does not reduce the size of input data, and it has to transfer a lot of intermediate records. Therefore, it is challenging for DistributedSort to collect data from all the places when the input data is large. When transferring data from one place to another, we make use of the RemoteArray object (Figure 5, Line 2). However, instances of user-defined types cannot be transferred by methods of RemoteArray object so far. We need to serialize them (Figure 5, Line 9) before offering them to the final reducer who will deserialize them before merging (Figure 5, Line 14).

```
1  store = new Array[Array[T]](place_num);
2  for(each place) remote_array(place_id) = new RemoteArray(store(place_id));
3  finish{
4    for(p in Place.places())
5      async at(p){
6        array = new Array[T](thread_num);
7        for(t in threads) read_and_sort(array(t));
8        output = merge_all_array(array);
9        stdoutput[char] = serialize(output);
10       Array.asyncCopy(stdoutput,remote_array(place_id));//offer data
11     }
12 }
13 data = deserialize(store);
14 result = do_merge_of(data);
```

Fig. 5. Serialization to transfer data in DistributedSort

4 Optimization

In this section, we present some optimizations to make the two applications more efficient.

4.1 Using C++ Native Calls to Process I/O

To our knowledge of the X10 standard library, the I/O performance decreases when processing large files. For C or C++ programming in Linux environment, the general approach to this problem is using the mmap() system call to map the file from disk directly to the user space address. However, there is no similar interface in X10 standard library.

Fortunately, X10 provides interfaces to invoke C++ or Java methods. In this way, we can easily call mmap() or similar methods to process the large files with high efficiency.

Figure 6 shows the processing time of reading files of 10MB, 20MB and 40MB size. The gap is significant between the X10 standard library and mmap(). The X10 standard library spends tens of seconds to read the files into the memory while mmap() only takes less than 1 second. Therefore, the interfaces to invoke the C++ native calls could be an alternative to deal with large files. We build a small library to encapsulate the mmap() system call for simplifying the invocation of C++ native calls and reusing the codes.

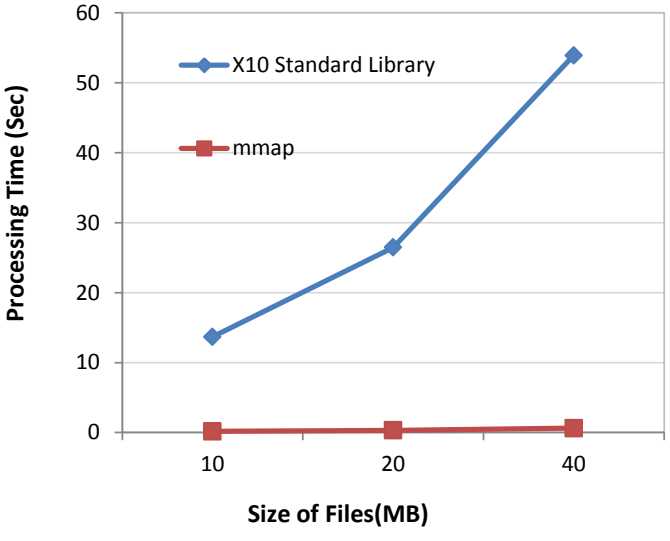


Fig. 6. Comparison of processing time between X10 standard library and mmap

4.2 Improving the Collecting-Finish Framework

We improve two phases of the default collecting-finish framework of X10 as following:

Merge Phase: First, we add filters to local data transfer, to avoid repeated copy of intermediate data on the same place. Second, we rewrite the reduce process to merge every two intermediate key/value pairs concurrently and recursively.

Sharing Data Among Places: In X10, remote data accesses requires some indirect mechanism. The standard collecting-finish framework uses deep copy to transfer data. Specifically, using *async at(p)* statement will cause deep copy, in which both primitive types and user-defined classes will be transferred.

However, when input data become large, deep copy will be very slow because the objects are transferred one by one and it may cause a descriptor error of sockets on Linux platform. Thus, we use the Array.asyncCopy method of X10 standard library to rewrite the data transferring process since Array.asyncCopy can copy a large chunk of memory at a time and provides an efficient data copy. However, it only supports arrays with default types. For user-defined types of data, it can only copy the reference of the object. In order to transfer data as a set of $\langle key, value \rangle$, we implement a simple serialization method to format the intermediate data to a byte array.

For WordCount, we simply use the default collecting-finish mechanism. However, in DistributedSort, which needs a large amount of data copy, we use the optimized implementation of collecting-finish to transfer large data.

Figure 7 shows the processing time of optimized collecting-finish framework and the original one. The suffix of "-opt" stands for our optimized version of collecting-finish. The suffix of "-std" stands for the standard library of X10. From the figure, we can see that the optimized implementation have a notable performance advantage over the original one. The speedup mainly comes from the merge phase, which involves lots of data transferring.

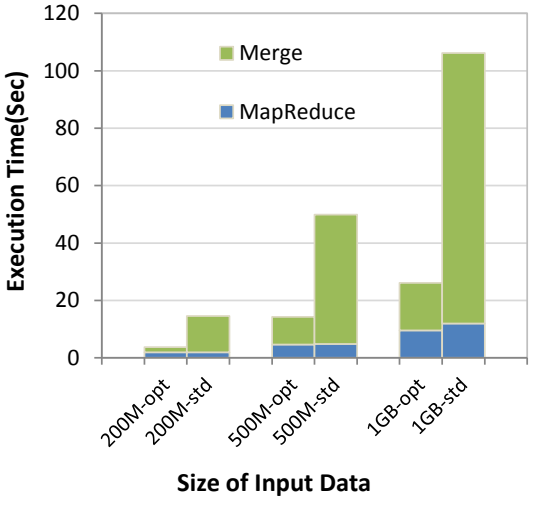


Fig. 7. Comparison of execution time between standard collecting-finish and the optimized one on a single node with 24 workers

5 Evaluation

In this section, we evaluate the performance of WordCount and DistributedSort applications implemented with X10 and compare with their Hadoop version.

5.1 Experiment Setup

The experiments are conducted on a small-scale cluster with 7 nodes. Each node has two AMD Opteron 12-core processors, 64 GB main memory. Every node is connected

to the same switch through a 1GB Ethernet link. The operating system is Debian 6.0 and the version of Hadoop is 0.20.1. We use X10 of version 2.1.1. The input size of the two applications varies from 200MB to 1GB. We evaluate the two applications both on a single node and on the cluster.

5.2 Overall Performance on a Single Node with Performance Breakdown

On a single machine, we evaluate the execution time of the two applications with different sizes of input data and different number of workers.

The execution time breakdowns of WordCount in Figure 8 shows the time spent in different phase of the WordCount application running in a single place. From the figure, we can see that the WordCount application has a good scalability. However, when the number of workers exceeds 9, the overhead of concurrency such as work scheduling and data access collision affect the continued improving of performance.

The MapReduce phase takes up most of the time. Currently, we maintain a sorted array to store the intermediate records generated by the map function. Each element contains a unique word and the number of its occurrences. To insert a new word, WordCount should search the word's position and then move all the words after that position backwards, to make room for this new word. The time for searching and moving depends on the number of unique words in the input file.

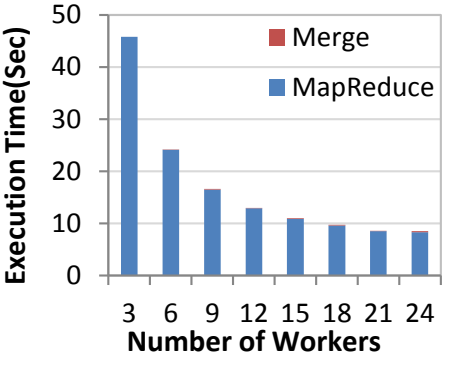


Fig. 8. Execution time breakdown of WordCount on a single node with different number of workers, using 1GB input data

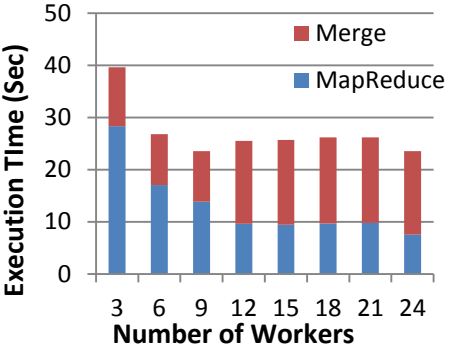


Fig. 9. Execution time breakdown of DistributedSort on a single node with different number of workers, using 1GB of input data

Figure 9 shows the performance trend according to the number of workers in DistributedSort. We can see when the number of workers increases to 9 it reaches quite a good performance. Then, until the worker numbers get to 24, the performance increases very little. This is because map tasks are parallelized well along with the increasing of the worker numbers. However, the reduce tasks, in fact, does not scale when the number of workers increases. By contrast, when the number of workers exceeds the required number of reduce tasks, an excessive amount of workers may cause frequent job stealing, which may slow down the execution of the program. Though the original launched

activity number of first phase is 24, there only needs 12 reducers to merge them, and during merging recursively, number of reducer decreases. Hence, when more workers are involved, the execution time will not necessary decrease. On the other hand, when the number of workers keeps increasing, the reduce time will not increase again, as shown in Figure 9. This helps X10 to keep its performance reasonably efficient without careful consideration on the worker number configuration.

5.3 Overall Performance on the Cluster and Comparison to Hadoop

On the cluster, we evaluate both the X10 MapReduce applications and those in Hadoop under the same environment. In X10, we use 7 places and there are 24 workers in each place. The Hadoop is configured with 7 nodes (one for master node and 6 for slave nodes), 144 map tasks (6 slaves and 24 tasks per slave) and 1 reduce task to merge all the output as in X10.

Figure 10 shows that the performance of WordCount in X10 significantly outperforms its Hadoop counterpart. The speedup mainly comes from the simpler task scheduling and less I/O for fault tolerance in X10. In Hadoop, a MapReduce job should be firstly submitted to the JobTracker running on the master node. Then, JobTracker further splits the job into map tasks and reduce tasks, and schedules them to slave nodes with available computing slots. In X10, when a job comes, it is just split and assigned in average to all workers. What's more, in Hadoop, the intermediate output produced by map tasks should be flushed to local disks before sending to reduce tasks for fault tolerance, which causes lots of I/O. Our X10 MapReduce applications take advantage of PGAS to share the data among all places so that they have reduce time less than the Hadoop ones.

Figure 11 shows comparison of DistributedSort between X10 and Hadoop on cluster. We can find that X10 still have efficiency advantages on concurrent computing, especially when data is small. The parallel primitives such as finish() statement detection, async() statement logic control, do not incurs too much overhead.

However, since X10 currently has not completed developing on its remote data transfer, we just implement a simple serializing and deserializing prototype for DistributedSort. So as the input data increases, the overhead of both serialization on map phase and deserialization on reduce phase grows quite quickly, which finally affects the whole computing efficiency of this application. We will try to improve the data transfer efficiency as future work.

6 Further Optimization Opportunities

Large Data Copy among Places: When copying a large amount of data of user-defined types, we suggest adding the serialization and deserialization of an array based on the current implementation of Array copy. We can serialize or even compress the data to be copied to the remote place and get a byte array that can be transferred asynchronous and fast by Array.asyncCopy. The remote place can access the data once the data is deserialized. This will be our future work. Meanwhile, since X10 is using socket-based data transfer, it may be beneficial to optimize the socket transfer structure of X10.

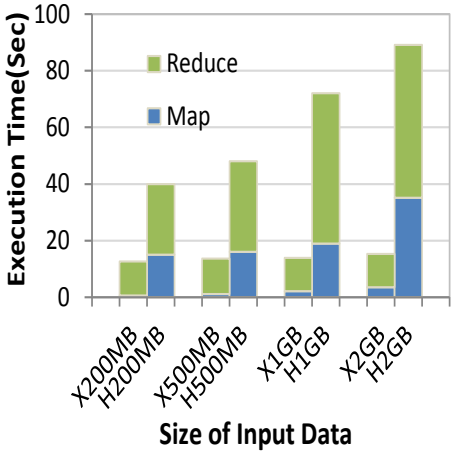


Fig. 10. Execution time of X10 WordCount on the cluster, as well as the time of one in Hadoop. The prefix 'X' represents X10, and 'H' represents Hadoop.

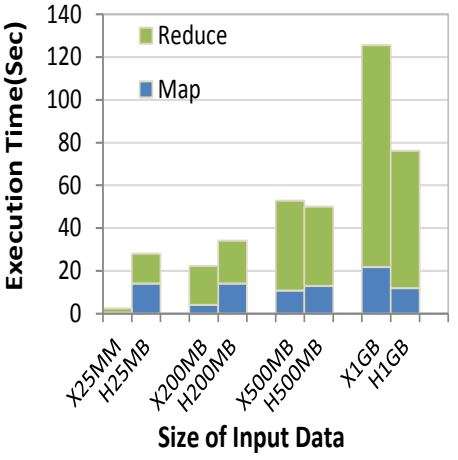


Fig. 11. Execution time of X10 DistributedSort on the cluster, comparing to the one in Hadoop. The prefix 'X' represents X10, and 'H' represents Hadoop.

Collecting-Finish: In the process of collecting-finish, there is only one or two workers involved in the reduce work while others are waiting for I/O, which is a waste of resources. Hence, other idle workers can be used to either for prefetching data or do data compression.

7 Related Work

X10 [3] is a new language under development. In recent years, much research has been done on X10 including optimizing X10 runtime, applying X10 in a variety of fields, etc.

A detailed tutorial about how to write X10 applications on modern architecture was provided by Sarkar et al. [5] and Murthy [6]. Saraswat et al. [7] developed a lifeline-based global load-balancing algorithm to extend the efficiency of work-stealing. Agarwal et al. [8] presented a framework that could statically establish place locality in X10 and offered an algorithm to eliminate runtime checks based on it. Agarwal et al. [9] proposed a lock-free scheduling mechanism for X10 with bounded resources.

The task creation and its termination detection in X10 will cause some overhead [10]. J. Zhao et al designed an efficient framework to eliminate it. Rajkishore Barik [11] proposed an optimized access way to the shared memory in parallel program both from low-level and high-level. The high-level optimizations are approaches to Side-Effect analysis and May-Happen-in-Parallel analysis in concurrent programs. The low-level ones mainly introduced the optimizations of compilation. The research of X10 also covers a Hierarchical Place Trees model which is as a portable abstraction for task parallelism and data movement [12], a compiling optimization of work-stealing in X10 [13], an optimized compiler and runtime of X10 improving the efficiency of executing instructions among places [14] and so on.

8 Conclusion and Future Work

MapReduce has been shown as a simple and efficient way to harness the massive resources of clusters.In this paper, we evaluated two MapReduce applications on the X10 platform, which was a member of languages based on partitioned global address space (PGAS). Though it is not an apple-to-apple comparison for X10 with Hadoop (as X10 currently does not support fault tolerance), our result showed that X10 is a powerful programming language to run MapReduce-style applications on clusters. By our study and analysis of X10 and its runtime, we eliminated some performance bottlenecks such as I/O processing. We also identified some remaining optimization opportunities. As future work, we are interested in improving the data transferring between places as well as the worker scheduling on multicore.

References

1. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51, 107–113 (2008)
2. Saraswat, V., Almasi, G., Bikshandi, G., Cascaval, C., Cunningham, D., Grove, D., Kodali, S., Peshansky, I., Tardieu, O.: The asynchronous partitioned global address space model. In: Proceedings of Workshop on Advances in Message Passing (2010)
3. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proc. OOPLSA, pp. 519–538 (2005)
4. Bialecki, A., Cafarella, M., Cutting, D., Omalley, O.: Hadoop: a framework for running applications on large clusters built of commodity hardware, `http://lucene.apache.org/hadoop`
5. Saraswat, V.A., Sarkar, V., von Praun, C.: X10: concurrent programming for modern architectures. In: Proc. PPoPP, pp. 271–271 (2007)
6. Murthy, P.: Parallel computing with x10. In: Proceedings of the 1st International Workshop on Multicore Software Engineering, pp. 5–6 (2008)
7. Saraswat, V.A., Kambadur, P., Kodali, S., Grove, D., Krishnamoorthy, S.: Lifeline-based global load balancing. In: Proc. PPoPP, pp. 201–212 (2011)
8. Agarwal, S., Barik, R., Nandivada, V.K., Shyamasundar, K., Varma, P.: Static detection of place locality and elimination of runtime checks. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 53–77. Springer, Heidelberg (2008)
9. Agarwal, S., Barik, R., Bonachea, D., Sarkar, V., Shyamasundar, R.K., Yelick, K.: Deadlock-free scheduling of x10 computations with bounded resources. In: Proc. SPAA, pp. 229–240 (2007)
10. Zhao, J., Shirako, J., Nandivada, V.K., Sarkar, V.: Reducing task creation and termination overhead in explicitly parallel programs. In: Proc. PACT, pp. 169–180 (2010)
11. Barik, R.: Efficient optimization of memory accesses in parallel programs (2009), `www.cs.rice.edu/~vsarkar/PDF/rajbarik_thesis.pdf`
12. Yan, Y., Zhao, J., Guo, Y., Sarkar, V.: Hierarchical place trees: A portable abstraction for task parallelism and data movement. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 172–187. Springer, Heidelberg (2010)
13. Raman, R.: Compiler support for work-stealing parallel runtime systems. M.S. thesis, Department of Computer Science, Rice University (2009)
14. Bikshandi, G., Castanos, J.G., Kodali, S.B., Nandivada, V.K., Peshansky, I., Saraswat, V.A., Sur, S., Varma, P., Wen, T.: Efficient, portable implementation of asynchronous multi-place programs. In: Proc. PPoPP, pp. 271–282 (2009)

Comparing High Level MapReduce Query Languages

Robert J. Stewart, Phil W. Trinder, and Hans-Wolfgang Loidl

Mathematical and Computer Sciences
Heriot Watt University

Abstract. The MapReduce parallel computational model is of increasing importance. A number of High Level Query Languages (HLQLs) have been constructed on top of the Hadoop MapReduce realization, primarily Pig, Hive, and JAQL. This paper makes a systematic performance comparison of these three HLQLs, focusing on *scale up*, *scale out* and *runtime* metrics. We further make a language comparison of the HLQLs focusing on conciseness and computational power. The HLQL development communities are engaged in the study, which revealed technical bottlenecks and limitations described in this document, and it is impacting their development.

1 Introduction

The MapReduce model proposed by Google [8] has become a key data processing model, with a number of realizations including the open source Hadoop [3] implementation. A number of HLQLs have been constructed on top of Hadoop to provide more abstract query facilities than using the low-level Hadoop Java based API directly. Pig [18], Hive [24], and JAQL [2] are all important HLQLs.

This paper makes a systematic investigation of the HLQLs. We investigate specifically, whether the HLQLs are indeed more abstract: that is, how much shorter are the queries in each HLQL compared with direct use of the API? What performance penalty do the HLQLs pay to provide more abstract queries? How expressive are the HLQLs - are they relationally complete, SQL equivalent, or even Turing complete? More precisely, the paper makes the following research contributions with respect to Pig, Hive, and JAQL.

A systematic performance comparison of the three HLQLs based on three published benchmarks on a range of Hadoop configurations on a 32 node Beowulf cluster. The study extends previous, predominantly single language studies. The key metrics we report are the *scale up*, *scale out* and *runtime* of each language. The performance baseline for each benchmark is a direct implementation using the Hadoop API, enabling us to assess the overhead of each HLQL (Section 6).

A language comparison of the three HLQLs. For each benchmark we compare the conciseness of the three HLQLs with a direct implementation using the Hadoop API using the simple *source lines of code* metric (Section 4.2). We further compare the expressive power of the HLQLs (Section 3.2).

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 58–72, 2011.

Our study is already impacting HLQL development. The HLQL development communities were engaged in the study, replicating our results and provided recommendations for each benchmark. A preliminary set of the results from the study [20] has been widely downloaded (1,200 downloads at present).

2 Related Work

2.1 MapReduce

Google propose MapReduce (MR) as "a programming model and an associated implementation for processing and generating large data sets" [8], and argue that many real world tasks are expressible using it. MR programmers are only required at a minimum to specify two functions: *map* and *reduce*. The logical model of MR describes the data flow from the input of *key/value* pairs to the *list* output:

```
Map(k1,v1) -> list(k2,v2)
Reduce(k2, list (v2)) -> list(v3)
```

The **Map** function takes an input pair and produces a set of intermediate key/-value pairs. The **Reduce** function accepts the intermediate *key2* and a set of values for that key. It merges these values to form a possibly smaller set of values [8]. The model has built-in support for fault tolerance, data partitioning, and parallel execution, absolving considerations like reliability and distributed processing away from the programmer.

In relation to High Performance Computing (HPC) platforms, [25] states that the MapReduce model is comparatively strong when nodes in a cluster require gigabytes of data. In such cases, network bandwidth soon becomes the bottleneck (leaving nodes idle for extended time periods) for alternative HPC architectures that use such APIs as Message Passing Interface (MPI), over shared filesystems on the network. MPI gives close control to the programmer who coordinates the dataflow via low-level routines and constructs, such as sockets, as well as higher-level algorithms for the analysis. In contrast, MapReduce operates only at the higher level, leaving the programmer to think in terms of functions on key and value pairs, and the data flow is implicit [25].

2.2 Hadoop

Hadoop [3] is an Apache open source MR implementation, which is well suited for use in large data warehouses, and indeed has gained traction in industrial datacentres at Yahoo, Facebook and IBM. The software stack of Hadoop is packaged with a set of complimentary services, and higher level abstractions from MR. The core elements of Hadoop however, are *MapReduce* - the distributed data processing model and execution environment; and the *Hadoop Distributed Filesystem* (HDFS) - a distributed filesystem that runs on large clusters. The HDFS provides high throughput access to application data, is suitable for applications that have large data sets, and the detection and recovery from faults is

a primary design goal of the HDFS. In addition, Hadoop provides interfaces for applications to move tasks closer to the data, as moving computation is cheaper than moving very large quantities of data [4].

Example MapReduce Application. Word count is a simple MapReduce application (Listing 1.1), commonly used for demonstration purposes [8]. The *map* function tokenizes a list of strings (one per line) as maps, and assigns an arbitrary value to each key. The reduce function iterates over this set, incrementing a counter for each key occurrence.

Listing 1.1. Pseudo Code: MapReduce Word Count

```
map( 'input.dat' ){
 Tokenizer tok <- file.tokenize();
   while (tok.hasMoreTokens){
     output(tok.next(),"1"); // list(k2,v2)
   }
}

reduce( word, values ){
 Integer sum = 0;
   while(values.hasNext()){
     sum += values.next();
   }
 output(word,sum); // list(v3)
}
```

3 High Level Query Languages

Justifications for *higher* level query languages over the MR paradigm are presented in [15]. It outlines the lack of support that MR provides for complex *N-step* dataflows, that often arise in real-world data analysis scenarios. In addition, explicit support for multiple data sources is not provided by MR. A number of HLQLs have been developed on top of Hadoop, and we review Pig [18], Hive [24], and JAQL [2] in comparison with raw MapReduce. Their relationship to Hadoop is depicted in Figure 1. Programs written in these languages are compiled into a sequence of MapReduce jobs, to be executed in the Hadoop MapReduce environment.

3.1 Language Query Examples

Pig - A High Level Data Flow Interface for Hadoop. *Pig Latin* is a high level dataflow system that aims at a sweet spot between SQL and MapReduce, by providing high-level data manipulation constructs, which can be assembled in an explicit dataflow, and interleaved with custom MR functions [15]. Programs written in *Pig Latin* are firstly parsed for syntactic and instance checking. The output from this parser is a logical plan, arranged in a directed acyclic graph,

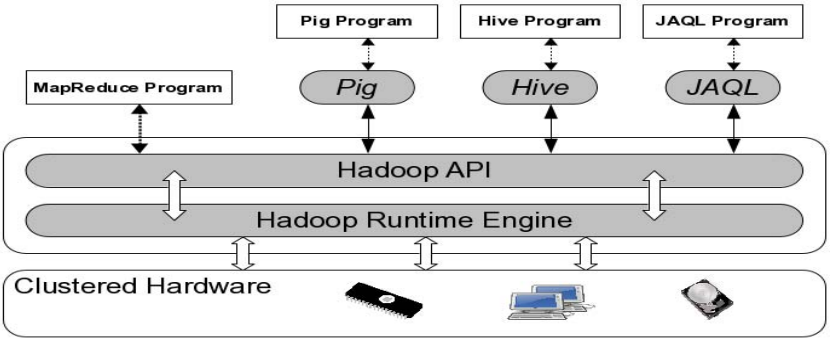


Fig. 1. HLQL Implementation Stack

allowing logical optimizations, such as *projection pushdown* to be carried out. The plan is compiled by a *MR compiler*, which is then optimized once more by a *MR optimizer* performing tasks such as early partial aggregation, using the MR *combiner* function. The MR program is then submitted to the Hadoop job manager for execution.

Pig Latin provides both simple scalar types, such as *int* and *double*, and also complex non-normalized data models. A *bytearray* type is supported, to facilitate unknown data types and lazy conversion of types, in addition to three collection types: *map, tuple* and *bag*. The Pig *word count* query is given in Listing 1.2.

Listing 1.2. Pig Word Count Benchmark

```
myinput = LOAD 'input.dat' USING PigStorage();
grouped = GROUP myinput BY \$0;
counts = FOREACH grouped GENERATE group,
         COUNT(myinput) AS total;
STORE counts INTO 'PigOutput.dat' USING PigStorage();
```

Hive - A Data Warehouse Infrastructure for Hadoop. *Hive QL* provides a familiar entry point for data analysts, minimizing the pain to migrate to the Hadoop infrastructure for distributed data storage and parallel query processing. Hive supports queries expressed in a SQL-like declarative language - HiveQL, which are compiled into MR jobs, much like the other Hadoop HLQLs. HiveQL provides a subset of SQL, with features like *from* clause subqueries, various types of *joins*, *group bys, aggregations* and *create table as select* all make HiveQL very SQL like.

Hive structures data into well-understood database concepts like tables, columns, rows, and partitions. It supports all the major primitive types: *integers, floats, doubles* and *strings*, as well as collection types such as *maps, lists* and *structs*. Hive also includes a system catalogue, a *metastore*, that contains schemas and statistics, which are useful in data exploration, query optimization and query compilation [23]. Just like with the Pig compilation process, Hive

includes a *query compiler*, which is a component that compiles HiveQL into a directed acyclic graph of MR tasks. The Hive *word count* query is given in Listing 1.3.

Listing 1.3. Hive Word Count Benchmark

```
CREATE EXTERNAL TABLE Text(words STRING)
LOCATION 'input.dat';
FROM Text INSERT OVERWRITE DIRECTORY 'HiveOutput.dat'
SELECT words, COUNT(words) as totals
GROUP BY words;
```

JAQL - A JSON Interface to MapReduce. *JAQL* is a functional data query language, which is built upon JavaScript Object Notation Language (JSON) [6]. JAQL is a general purpose data-flow language that manipulates semi-structured information in the form of abstract JSON values. It provides a framework for reading and writing data in custom formats, and provides support for common input/output formats like CSVs, and like Pig and Hive, provides operators such as *filtering, transformations, sort, group bys, aggregation,* and *join* [2].

JSON supports atomic values like numbers and strings, and has two container types: arrays and records of name-value pairs, where the values in a container are arbitrary JSON values. Databases and programming languages suffer an *impedance mismatch* as both their computational and data models are so different [1]. As the JSON model provides easy migration of data to and from some popular scripting languages like Javascript and Python, JAQL is extendable with operations written in many programming languages because JSON has a much lower impedance mismatch than XML for example, yet much richer datatypes than relational tables [2]. The JAQL *word count* query is given in Listing 1.4.

Listing 1.4. JAQL Word Count Benchmark

```
$input = read(lines('input.dat'));
$input -> group by $word = $
into { $word, num: count($) }
->write(del('JAQLOutput.dat',{fields:[''words'',''num'']}));
```

3.2 HLQL Comparison

Language Design. The language design motivations are reflected by the contrasting features of each high level query language. Hive provides Hive QL, a SQL like language, presenting a declarative language (Listing 1.3). Pig by comparison provides Pig Latin (Listing 1.2), a dataflow language influenced by both the declarative style of SQL (it includes SQL like functions), and also the more procedural MR (Listing 1.1). Finally, JAQL is a functional, higher-order programming language, where functions may be assigned as variables, and later evaluated (Listing 1.4). In contrast, Pig and Hive are strictly evaluated during the compilation process, to identify type errors prior to runtime.

Computational Power. Figure 2 illustrates the computational power of Pig, Hive, and JAQL, the MR model, and the MapReduceMerge model [27]. MapReduceMerge was designed to extend MR, by implementing relational algebra operators, the lack of which signifies that MR itself is not relationally complete. Indeed, MR is a simple model for applications to be encapsulated for computation in distributed environments.

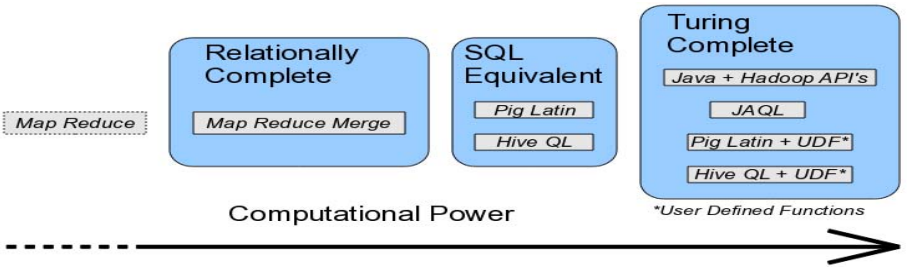


Fig. 2. Computational Power Comparison

Relational Completeness. Relational algebra is a mathematical notation that defines relations in standardized algebraic syntax which are combined to create the operators and functions required for relational completeness [7]. The structured query language, SQL, is regarded as relationally complete as it provides all operations in relational algebra set, and in addition offers a set of aggregation functions, such as *average, count*, and *sum*.

Turing Completeness. A language that contains conditional constructs; recursion capable of indefinite iteration; and a memory architecture that emulates an infinite memory model is said to be Turing Complete.

User Defined Functions. Pig, Hive and JAQL are all extendable with the use of user defined functions (UDF). These allow programmers to introduce custom data formats and bespoke functions. They are always written in Java, a Turing complete language.

4 Conciseness

4.1 Application Kernels

The HLQLs are benchmarked using three applications: two that are canonical and published, and a third that is a typical MR computation. The word count in Figure 3 is the canonical MR example [10]. The dataset join in Figure 4 is another canonical example, with published implementations for Java, Pig, Hive and JAQL [9,26,11,16]. The web log aggregation example in Figure 5 is a new benchmark that is typical of webserver log manipulation. Each can be described concisely in SQL, as:

```
SELECT words, COUNT(words) as totals
GROUP BY words;
```

Fig. 3. Word Count

```
SELECT t1.*
FROM TABLE1 t1
JOIN TABLE2 t2
  ON (t1.field = t2.field);
```

Fig. 4. Dataset Join

```
SELECT userID, AVG(timeOnSite) as averages, COUNT(pageID)
GROUP BY userID;
```

Fig. 5. Web Log Processing

4.2 Comparative Code Size

Table 1 shows the source lines of code count to satisfy the requirements used in the word count, join, and web log processing applications.

Table 1. Source Lines of Code Comparison

	Java	Pig	*Pig/Java Ratio*	JAQL	*JAQL/Java Ratio*	Hive	*Hive/Java Ratio*
Word Count	45	4	8.9%	6	13.3%	4	8.9%
Join	114	5	4.4%	5	4.4%	13	11.4%
Log Processing	165	4	2.4%	3	1.8%	11	6.7%
Mean Ratio	(100%)		**5.2%**		**6.5%**		**9%**

The aim of the high level languages Pig, Hive and JAQL is to provide an abstract data querying interface to remove the burden of the MR implementation away from the programmer. However, Java MR applications requires the programmer to write the *map* and *reduce* methods manually, and hence require program sizes of a magnitude of at least 7.5 (Table 1). It would appear that conciseness is thus achieved for the equivalent implementations in the HLQLs, and Section 6 will determine whether or not programs pay a performance penalty for opting for these more abstract languages. Refer to further language discussion, in Section 7.

5 Performance Experiment Design

This section outlines the runtime environment used for the performance comparisons, justifies the number of reducers used in each experiment, and explains the importance of key distribution for MapReduce applications.

5.1 Language Versions, Hardware and Operating Systems

The experiments utilize Pig version 0.6, Hive version 0.4.0, an *svn* snapshot of JAQL (r526), and Hadoop version 0.20.1. The benchmarks were run on a cluster

comprising of 32 nodes running Linux CentOS 5.4; Kernel 2.6.18 SMP; Intel Pentium 4, 3Ghz dual core, 1MB cache, 32bit; 1GB main memory; Intel Pro 1Gbps Full Duplex; HDD capacity 40GB.

5.2 Reducer Tasks

Guidelines have been outlined for setting a sensible value for the number of reducers for Hadoop jobs. This value is determined by the *number of Processing Units* (nodes) **PU**s in the cluster, and *the maximum number of reducers* per node **R**, as $\mathbf{PU} \times \mathbf{R} \times \mathbf{0.9}$ [17].

5.3 Key Distribution

A potential bottleneck for MR performance can occur when there exists a considerable range in the number of values within the set of keys generated by the *map* function (see Section 2.1). In Section 6.1, we measure the performance of each language when processing input data with skewed key distribution. The skewed dataset was generated using an existing package [22], developed by the Pig language design team at Yahoo.

6 Performance Results

The application kernels outlined in Section 4.1 are used in this section to measure the language performance for scale-up, scale-out and runtime. It also shows the effect on runtime when tuning the number of reducers, for each language. A full specification of the Hadoop runtime parameters for all of the experiments that are described, can be found in Section 3.2 of [19].

6.1 Scaling Input Size

To measure the *scale-up* performance of each language, the size of the cluster was fixed at *20 PU's*. Scaling with uniform key distribution computation is shown in figures 6 and 7, and with a skewed key distribution computation is shown in figures 8 and 9.

A prominent feature of these measurements is that total runtime increases with input size. Two trends emerge, such that JAQL and Pig achieve similar scale up performance, though both Hive and Java perform considerably better. In the web log processing with the smallest input size, JAQL is the most efficient processing engine (Figure 7), achieving the lowest runtime of all of the languages - 26% quicker than the lower level Java implementation.

Skewed Keys. The word count and join applications were executed on datasets with skewed key distribution, in Figures 8 and 9. Both JAQL and Java performances are impaired by the skewed key distribution in the *join* application, scaling up less well. On feeding preliminary join results (Figure 9) back to the

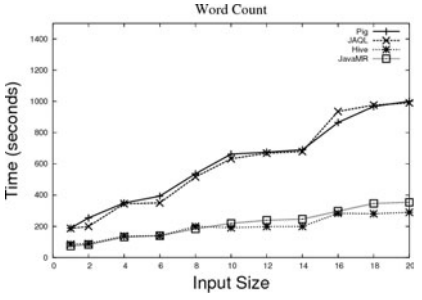


Fig. 6. Word Count Scale Up - Uniform Distribution

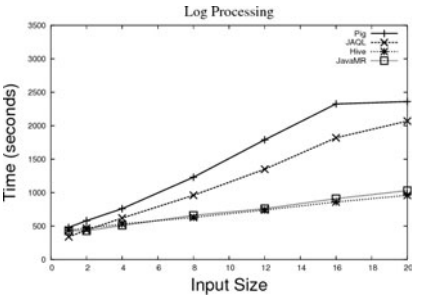


Fig. 7. Web Log Processing Scale Up - Uniform Distribution

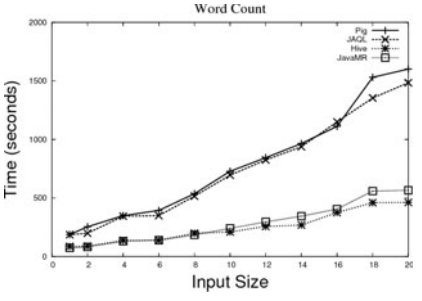


Fig. 8. Word Count Scale Up - Skewed Distribution

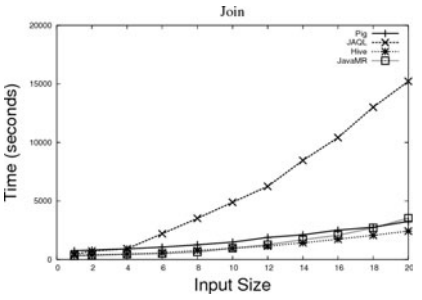


Fig. 9. Dataset Join Scale Up - Skewed Distribution

JAQL development team, a problem with the JAQL delimited output method was identified, and subsequently fixed [21]. Both Pig and Hive appear to handle the skewed key distribution more effectively. In Figure 9, where input size is *x20*, both Pig and Hive outperform the Java implementation. A complete result set from the skewed key runtime experiments can be found at [19].

6.2 Scaling Processing Units

To measure the *scale-out* performance of each language, the size of computation was fixed, and the number of worker nodes was increased from 1 to 20. The results of these *scale-out* benchmarks are depicted in Figures 10 and 11.

Beyond 4 PU's, there is no apparent improvement for JAQL, whereas Pig, Hive and Java are all able to utilize the additional processing capacity up to 16 PU's, at which point, no further cluster expansion is beneficial to runtime performance.

A fundamental contrast between the *join* and the *web log processing* application, in Figures 10 and 11 respectively, is the input size for the two applications.

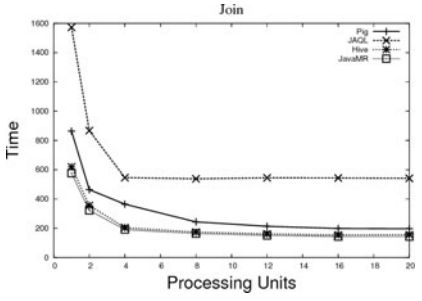


Fig. 10. Runtime for Dataset Join

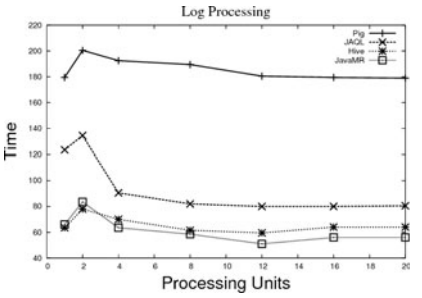


Fig. 11. Runtime for Web Log Processing

As a result, less work is required by every run of the *web log processing* application, and more for the *join* application. These results clearly illustrate one common design challenge for parallel systems - In the *join* application (Figure 10), a sufficient workload eluded to a diminished runtime, shifting from using 1 PU, to 2 - a trivial case study. However, there was only modest computational requirements for the *web log processing* application, Figure 11. It appears that, as a consequence, the communication and coordination overheads associated with work distribution (2 or more PU's) is detrimental to overall runtime performance.

6.3 Proportional Scale Out of Input Size and Processing Units

In this experiment, the required computation (**C**) is increased in proportion with the number of PU's **P** in the cluster, with a multiplying factor **m**. Ideal performance would achieve no increase in the computation time **T** as **m** increases.

$$\mathbf{T = mC \ / \ mP}$$

In reality however, this is not attainable, as communication and task coordination costs influence overall runtime, hindering runtime performance.

Whilst Java achieves the quickest runtime for the *join* operation when *scaling out* (Figure 12), Pig and Hive achieve performance parity with Java for the *relative* scale out performance, Figure 13. The exception is JAQL, which scales less well.

6.4 Controlling Reducers

All MR jobs are split into many *map* and *reduce* tasks. The computational granularity - the computation size per reduce task, may have a significant influence of performance.

Whilst the number of *map tasks* for a MR job is calculated by the Hadoop runtime engine, there is a degree of flexibility for the number of *reduce tasks*

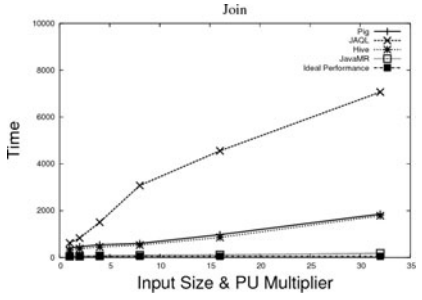


Fig. 12. Dataset Join Scale Out

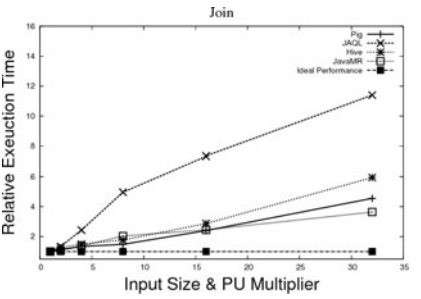


Fig. 13. Relative Dataset Join Scale Out

associated with a job. When these experiments were conducted in May 2010, Hive, Pig and the Hadoop Java API provided an additional performance tuning parameter to specify the number of reducers to be used, whilst JAQL did not. Figures 14 and 15 are annotated where the reducers parameter is set at 18, in accordance with the formula in Section 5.2.

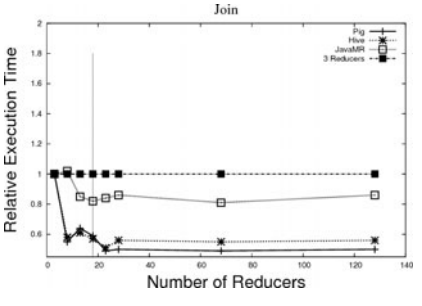


Fig. 14. Controlling Reducers: Dataset Join

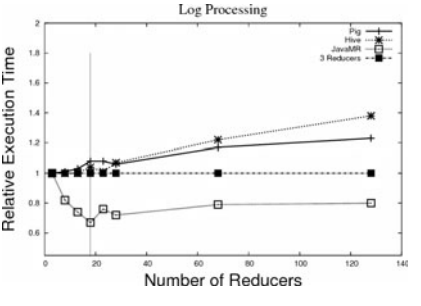


Fig. 15. Controlling Reducers: Log Processing

A consequence of too-fine granularity is in an increase in runtime as the reduce tasks parameter increases to 128, shown in Figure 15. In addition, poor runtime performance is seen where the *reduce tasks* parameter is set well below the recommended value, 18. The *join* application in Figure 14 shows that despite the parameter guidance value of 18, *Pig* and *Java* achieve quickest runtime at 23 reducers, after which, performance gradually degrades.

In summary, the additional expressive power of controlling the number of reducer tasks can optimize performance by as much as 48% (see Figure 14) compared to the default setting, set by the default Hadoop configuration, of just 1 reducer. In addition, the results illustrate that the guideline for the value of this parameter (detailed in Section 5.2) is approximately optimal for this *"control knob"* in achieving the quickest runtime for the application.

7 Language Discussion

A common feature of Pig, Hive and JAQL is that they can be extended with special-purpose Java functions. This means that they are not limited to the core functionality of each language, and the computational power is increased with the use of such user defined functions, as detailed in Section 3.2.

Pig is the most concise language, with an average ratio (relative to Java) of just 5.2%, as shown in Table 1. Pig has built-in optimization for joining skewed key distribution, which outperforms data handling found in a non-adaptive Java MR join application, shown in Figure 9. The discussion of tuning reduce tasks in Section 6.4, in Figure 14, shows that Pig takes advantage of an increasing number of reduce tasks, and handles an undesirably high level of specified reducer tasks comparatively well in Figure 15.

Hive was the best performer for the scale-up, scale-out and speed-up experiments for each of the three benchmarks, and throughout these experiments, overall runtime was only fractionally slower than Java. Like Pig, Hive took advantage of tuning the reducers parameter in the join application, in Figure 14. Both Pig and Hive are shown to have SQL equivalent computational power[1].

JAQL is a lazy higher-order functional language and is Turing Complete, whereas Pig Latin and Hive QL are not. Throughout the benchmark results, JAQL does not compete with the runtime of Hive or Java. However, it does achieve a speed-up of approximately 56% for the join benchmark, shown in Figure 10. The runtime improvements through controlling the number of reducers (Section 6.4) were not measured for JAQL at the date of experiment execution (May, 2010), though this feature has subsequently been added to the language [5]. This highlights one challenge for comparative studies such as this report - these languages are relatively new, and are moving targets when attempting to construct fair and relevant performance and feature comparisons.

One particularly apt challenge when writing comparative studies such as this, is that each project - each HLQL, is a "moving target". Active development continues to evolve each language, with new features being added with each release. Pig 0.8, for example, was released on the 17th December 2010, and provides support for UDFs in scripting languages, and also a safeguard against the omission of the "number of reducers" tuning parameter. Given such an omission, Pig uses its own heuristic to set this value, to avoid inefficient runtimes discussed in Section 6.4. Pig version 0.8 also introduces a *PigStats* utility, enabling "Pig to provide much better visibility into what is going on inside a Pig job than it ever did before" [12]. This utility would be an invaluable tool for further comparisons, and tuning, of Pig performance for more opaque and complex programs. Likewise, Hive 0.7 was released on the 29th March 2011, which adds data indexing, a

[1] With the minor exception that Pig does not currently provide support for arbitrary *theta* joins.

concurrency model, and an authentication infrastructure, amongst other things [13].

The complimentary nature of these HLQLs is discussed in [14], by a member of the Pig development team. The analogy outlines three separate tasks involved with data processing: *data collection*; *data preparation*, performed in "data factories"; and *data presentation*, performed in "data warehouses", and this study has focused on the latter two. The data engineers at Yahoo appear to believe that Pig, with its support for pipelines and iterative processing, is a good solution for data preparation. Similarly Hive, with its support for ad-hoc querying and its query support for business intelligence tools, is a natural fit for data presentation. The conclusion in [14] is that the combination of Pig and Hive therefore provides a complete data processing toolkit for Hadoop.

8 Conclusion

This report has presented a comparative study of three high level query languages, that focus on data-intensive parallel processing, built on top of the MapReduce framework in Hadoop. These languages are Pig, Hive and JAQL, and a set of benchmarks were used to measure the scale-up, scale-out and runtime of each language, and ease of programming and language features were discussed.

Hive was shown to achieve the quickest runtime for every benchmark, whilst *Pig* and *JAQL* produced largely similar results for the scalability experiments, with the exception of the *join* application, where *JAQL* achieved relatively poor runtime. Both *Hive* and *Pig* have the mechanics to handle skewed key distribution for some SQL like functions, and these abilities were tested with the use of skewed data in the *join* application. The results highlight the success of these optimizations, as both languages outperformed Java when input size was above a certain threshold.

The report also highlighted the abstract nature of these languages, showing that the source lines of code for each benchmark is much smaller than the Java implementation for the same benchmark, by a factor of at least 7.5. *Pig* and *Hive* enable programmers to control the number of reducers within the language syntax (as of May 2010), and this report showed that the ability to tune this parameter can greatly improve the runtime performance for these languages. JAQL was shown to be the most computationally powerful language, and it was argued that JAQL was *Turing Complete*.

References

1. Atkinson, M.P., Buneman, P.: Types and persistence in database programming languages. ACM Comput. Surv. 19(2), 105–190 (1987)
2. Beyer, K.S., Ercegovac, V., Krishnamurthy, R., Raghavan, S., Rao, J., Reiss, F., Shekita, E.J., Simmen, D.E., Tata, S., Vaithyanathan, S., Zhu, H.: Towards a scalable enterprise content analytics platform. IEEE Data Eng. Bull. 32(1), 28–35 (2009)

3. Borthakur, D.: The Hadoop Distributed File System: Architecture and Design (2007), http://www.hadoop.apache.org
4. Borthakur, D.: The Hadoop Distributed File System: Architecture and Design. The Apache Software Foundation (2007)
5. code.google.com/p/jaql. Jaql developers message board, http://groups.google.com/group/jaql-users/topics
6. Crockford, D.: The application/json media type for javascript object notation (json). RFC 4627 (Informational) (July 2006)
7. Date, C.J.: An Introduction to Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston (1991)
8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
9. The Apache Software Foundation. Hadoop — published java implementation of the join benchmark, http://goo.gl/R4ZRd
10. The Apache Software Foundation. Hadoop — wordcount example, http://wiki.apache.org/hadoop/WordCount
11. The Apache Software Foundation. Hive — language manual for the join function, https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Joins
12. The Apache Software Foundation. Pig 0.8 — release notes (December 2010), http://goo.gl/ySUln
13. The Apache Software Foundation. Hive 0.7 — release notes (March 2011), http://goo.gl/3Sj67
14. Gates, A.: Pig and hive at yahoo (August 2010), http://goo.gl/OVyM1
15. Gates, A.F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S.M., Olston, C., Reed, B., Srinivasan, S., Srivastava, U.: Building a high-level dataflow system on top of map-reduce: the pig experience. In: Proc. VLDB Endow., vol. 2, pp. 1414–1425 (August 2009)
16. IBM. Jaql — language manual for the join function, http://code.google.com/p/jaql/wiki/LanguageCore#Join
17. Murthy, A.C.: Programming Hadoop Map-Reduce: Programming, Tuning and Debugging. In: ApacheCon US (2008)
18. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD 2008: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1099–1110. ACM, New York (2008)
19. Stewart, R.J.: Performance and programmability comparison of mapreduce query languages: Pig, hive, jaql & java. Master's thesis, Heriot Watt University, Edinburgh, United Kingdom (May 2010), http://www.macs.hw.ac.uk/~rs46/publications.php
20. Stewart, R.J.: Slideshow presentation: Performance results of high level query languages: Pig, hive, and jaql (April 2010) http://goo.gl/XbsmI
21. JAQL Development Team. Email discussion on jaql join runtime performance issues. private communication (September 2010)
22. Pig Development Team. Pig DataGenerator, http://wiki.apache.org/pig/DataGeneratorHadoop
23. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive - a petabyte scale data warehouse using hadoop. In: ICDE, pp. 996–1005 (2010)

24. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a map-reduce framework. In: Proc. VLDB Endow., vol. 2(2), pp. 1626–1629 (2009)
25. White, T.: Hadoop — The Definitive Guide: MapReduce for the Cloud. O'Reilly, Sebastopol (2009)
26. Yahoo. Pigmix — unit test benchmarks for pig, http://wiki.apache.org/pig/PigMix
27. Yang, H.-c., Dasdan, A., Hsiao, R.-L., Parker, D.S.: Map-reduce-merge: simplified relational data processing on large clusters. In: SIGMOD 2007: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 1029–1040. ACM, New York (2007)

A Semi-automatic Scratchpad Memory Management Framework for CMP

Ning Deng, Weixing Ji*, Jaxin Li, and Qi Zuo

Beijing Institute of Technology, Beijing, 100081, China
pass@bit.edu.cn

Abstract. Previous research has demonstrated that scratchpad memory(SPM) consumes far less power and on-chip area than the traditional cache. As a software managed memory, SPM has been widely adopted in today's mainstream embedded processors. Traditional SPM allocation strategies depend on either the compiler or the programmer to manage the small memory. The former methods predict the frequently referenced data items before real running by static analysis or profiling, whereas the latter methods require the programmer to manually allocate the SPM space. As for the dynamic heap data allocation, there is no mature allocation scheme for multicore processors with a shared software-managed on-chip memory. This paper presents a novel SPM management framework, for chip multiprocessors (CMP) featuring partitioned global address space (PGAS) SPM memory architecture. The most frequently referenced heap data are maintained in the SPM. This framework mitigates the SPM allocation problem by leveraging the programmer's hints to determine the data items allocated to the SPM. The complex and error-prone allocation procedure is completely handled by an SPM management library (SPMMLIB) without programmer's conscious. The performance is evaluated in a homogenous UltraSPARC multiprocessor using PARSEC and SPLASH2 benchmarks. Experimental results indicate that, on average, the energy consumption is reduced by 22.4% compared with the cache memory architecture.

1 Introduction

Cache is the most popular on-chip memory in mainstream desktop processors, whereas scratchpad memory (SPM) is an alternative or complementary to the traditional cache in many embedded processors. SPM is a general term of the on-chip SRAM without a tag logic. Different from the cache's tag comparison mechanism, SPM is usually mapped into the unified address space of the main memory. The programmer can access the small on-chip memory using its continuous logic address. The most significant advantage of SPM is its lower power and higher integrity compared with the cache. According to the statistics in [3], SPM costs up to 40% less energy and 34% less area than cache. Unlike the cache's automatic management through hardware, SPM is mainly managed by the compiler or the programmer via software. Additionally, SPM access doesn't need to consider the random miss/hit overhead, thus making the prediction of the execution

* Corresponding author.

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 73–87, 2011.

much easier. This is an important feature for systems whose real-time constraints are highly regarded.

Programs running on the application-specific platform are tied during manufacturing and will never be changed henceforth. Thus, SPM management on the application-specific processor is comparatively easy because the program behavior is simple and permanent. In general-purpose processors, the memory reference behavior is more complex and is greatly affected by the outside input or execution environment. In most of the embedded general-purpose systems, SPM is explicitly managed by the compiler or is manually controlled by the programmer. In the compiler-directed methods, profiling knowledge or code static analysis is the most popular techniques to predict the frequently accessed data items. Almost all the previous methods share the common goal of predicting the most profitable hot data items and allocating them in the SPM to save some energy and execution time.

Specifically, as the recent revolutionary changes brought by personal customer electronic products, such as smart-phone and tablet PCs, more and more commercial MPSoC embedded processors have emerged to provide richer computational resources while maintaining a longer battery life. Embedded applications are becoming more general purpose, which makes SPM management an attractive and challenging topic for the development of embedded general-purpose applications. In commercial MPSoC architectures, such as IBM CELL [13] and NVIDIA Fermi [1], SPM and its variants have been embedded into practical processors to improve their performance. CELL BE processor has a local memory in every SPE , which transfers data with PPE through DMA. The local memory allocation, data transfer, and even the local memory coherence are explicitly managed by the programmer through SDK, which makes the programming on CELL laborious. In the NVIDIA Fermi architecture, each SM has 64 KB of on-chip memory, which can be configured into cache plus SPM according to the application's behavior. Generally, a programmer-oriented SPM management requires the programmer to have intense understanding of the architecture and the program's behavior, which obviously makes the programming tough and error-prone. However, the completely automatic SPM management without the programmer's awareness may also incur inaccurate hot data allocations and an undesirable performance degradation.

This paper proposes an SPM management framework, called the SPM management library (SPMMLIB), for tile-based CMP. SPMMLIB can serve the basic SPM management functions in a parallel programming model, including the allocation, deallocation, and distributed data layout in SPM. When allocating an SPM space, the programmer provides the expected data for SPM allocation. The annotated candidates are then processed in the charge of the SPMMLIB. The framework relieves the programmer's burden and makes the cumbersome operations invisible to the programmer. SPMMLIB aims at a series of CMP features in the partitioned global address space (PGAS) on-chip memory model, where the logically shared memories are physically distributed at multiple cores. In multithreading programs, the shared SPM data may incur references from threads running on the remote cores, namely, the remote accesses, leading to a much higher reference overhead with regard to the local SPM references. SPMMLIB considers the data-thread affinity when conducting the SPM allocations. Once the programmer annotates the expected partitioned data, the framework

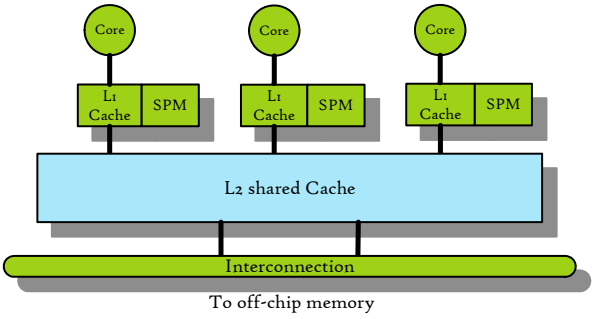


Fig. 1. Architecture of the PGAS SPM system

automatically partitions the data structures into small slices and allocates them to multiple cores running the data-aware threads. In sum, this paper makes the following contributions: (1) compare several on-chip memory configurations through an experiment and provide a semi-automatic SPM management framework for the tiled CMP that features the PGAS memory hierarchy; (2) support both the normal SPM allocation and an affinity-aware data partition allocation for the multithreading programs to enhance its locality; and (3) implement and evaluate the performance of the SPMMLIB in the general-purpose multicore architecture using the full-system simulation.

2 The Target Architecture

There are two alternatives when constructing a multicore SPM architecture. The first alternative is called the limited local memory(LLM). Local memory in the CELL BE processor is a typical representative [2] of this architecture. Every single SPE in the CELL processor holds a private SPM. The SPE can only reference data items in its local memory, and a remote access to the peer SPE's local memory is invalid. Once the SPE needs to access a data location in the remote local memory or in the off-chip DRAM, a data copy is first transferred through the DMA explicitly managed by the programmer to the SPE's local memory. Apparently, the data copies residing in the multiple SPEs' local memory may incur the coherence problem, and there is no hardware mechanism to maintain the coherence among the different local memories. In the second CMP-based SPM architecture, as shown in Fig.1, each processor tile integrates some SPM, which is physically distributed at the processor cores but shares the same logic address space. The overall SPM is organized as a PGAS system. In this model, multiple cores share the same SPM address space, and all data items are unique. Thus, no coherence protocol is required to protect the modification to the multiple data copies. However, the distributed SPM in multiple tiles leads to an non-uniform SPM access. The access requests to the local SPM spend less cycles (about 1-2 cycles) than the remote SPM references, which is about an order of magnitude larger than the local accesses. Similar to the non-uniform cache accesses(NUCA) in the last-level cache in the CMP, the accesses to the PGAS SPM are considered as the non-uniform SPM accesses (NUSA).

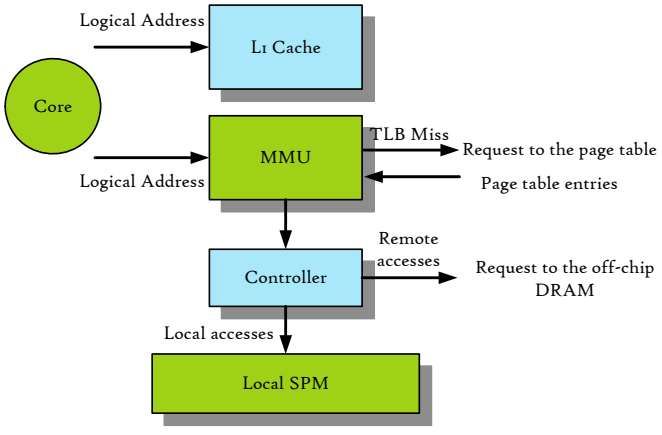


Fig. 2. Datapath of the SPM system

We select the second architecture as the target architecture in our study and implement the CMP-based SPM model in a quad-core UltraSparc multicore environment. However, it should note that our method is qualified for all MPSoC/CMP models with a PGAS SPM feature. In previous SPM architectures, some research only adopt SPM to replace cache as the on-chip memory while more select the SPM/cache combinations. A reasonable explanation to this is that cache delivers better performance in irregular accesses while SPM is suitable in satisfying regular frequent accesses. An experimental result shows the performance disparity in Section 4. We consider the use of SPM as an on-chip memory is not merely for reducing the execution cycles but leveraging its innate energy advantage over cache.

As shown in Fig.2, the memory requests issued by the core are accesses to either cache or the memory. The issued logical address is sent to L1 cache and the MMU/TLB at the same time, and the virtual addressed cache enables that the cache lookups can be conducted at the same time as the TLB/MMU lookups to save some cycles. Once the data is hit in the L1 cache, the processor core gets the data and continue the next instruction execution. Otherwise, if there is a L1 cache miss, the logical address is translated into its corresponding physical address by MMU. The physical address is then compared with a predefined address register to determine if the requested data item is located in the SPM. Once an SPM access is confirmed, the memory controller will further classify it as either a local SPM access or a remote one according to its physical address. All requests to the off-chip memory are directed as they are processed in the cache architecture.

3 THE SPMMLIB Runtime System

3.1 Motivation

Heap data refer to the dynamic data items explicitly controlled by the programmer and are dynamically allocated at runtime. The growing direction of the heap is from the

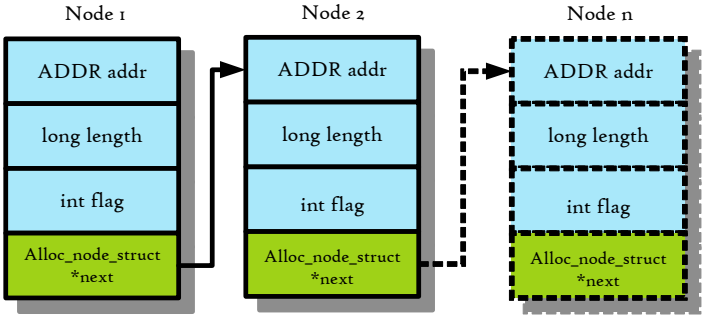


Fig. 3. List structure inside the SPMMLIB

small memory address to the large memory address. Current Linux kernel patches have tried to support the Samsung U600 (ARMv7 Instruction Set) 's TCM [25], which is a form of SPM in the ARM architecture. However, system-level support for the SPM in CMP has not been thoroughly studied yet.

SPMMLIB is expected to provide a set of simple interfaces that hide the complex SPM management operations. The decision of what data should be allocated is made by the programmer, who is expected to know the access patterns and the thread-data affinity when coding. Compared with the previous SPM management methods, our framework needs only the simplest hint to decide the SPM data candidates. The SPMMLIB takes the responsibility of handling the tedious and error-prone work henceforth. In our study, SPMMLIB is implemented as a software dynamic library for simplicity, but it is easy to be embedded into the future OS supporting the software-managed memory.

3.2 Inside the SPMMLIB

From the perspective of SPMMLIB, the SPM is maintained as a consequence memory space constructed by memory blocks of the same size. We use a bitmap and an allocation list in SPMMLIB to manage the SPM space as a whole. A bitmap called SPM_PHYS_POS[$BITMAP\_SIZE$] is adopted to record the available physical positions in SPM. SPM_PHYS_POS[i] = 0 indicates that the *ith* block in the whole SPM block sequence is not allocated; otherwise, SPM_PHYS_POS[i] = 1 means that the *ith* block has been allocated and in use. When allocating an SPM space, the bitmap is scanned to determine the first index of one of the continuous spaces that satisfy the allocation size. After the SPM space is free, the related bitmap positions are also reset to identify the availability for future use. The selection of the block size affects the performance of our method. A too large SPM block size has a greater chance of incurring a low SPM utilization when the requested memory size does not fit the multiple of the block size, whereas a too small SPM block size results in a large bitmap, which needs more space and higher scanning overhead. In the UltraSPARC architecture, the minimum page size is 8KB. We select 1KB as the block size, which is smaller than the plain memory page size and is suitable for SPM maintenance.

As illustrated in Fig.3, a list data structure is constructed to record information. Once an SPM allocation is conducted, a new node is created and linked at the tail of the allocation list. The node structure contains some basic records of this allocation. *Addr*, which is the first field in the node structure, represents the start address returned by this allocation, and the *length* records the SPM size of this allocation. SPMMLIB supports two kinds of memory allocation, each of which includes more than one allocation situations. Thus, the field named $flag$ identifies all these different allocation situations. The detail of each situation is explained in Section 3.3. The last field of the node is a pointer to the next node.

When conducting an allocation, the bitmap is first scanned to find a suitable position. A new allocation node is then created and inserted at the tail of the list. Information such as the start address, length, allocation situation is kept inside the allocation list node. During a free operation, the list is linearly scanned to search a node of the equal start address to the given freed address. Then, the related positions in the bitmap is reset according to the length kept in the allocation list.

3.3 SPMMLIB Application Programming Interface

void spm_malloc_init(void) initializes the bitmap and the list to prepare for the SPM allocation operations. When the SPMMLIB is embedded into the future OS, the initialization system call also takes the responsibility of checking if there is SPM in the system and what the capacity is. Such information is hard-coded in our implementation.

void *spm_normal_malloc(long bytenum) manages the SPM as a whole memory space, and the allocation is dealt with linearly. This interface allocates an SPM space, whose length equals $(bytenum + BLOCK\_SIZE - 1)/BLOCK\_SIZE * BLOCK\_SIZE$. Once the available SPM can satisfy the request, it returns the start address according to the bitmap index and sets the flag to NORMAL_SPM; otherwise, the allocation is handled by the *malloc*() system call, and the flag is set to NORMAL_MEM. The bitmap is set when the allocation is completed.

void **spm_distributed_malloc(long bytenum) ensures that the memory request is satisfied among all the processor cores. In an n-core multicore system, this interface equally allocates the on-chip memory space in the SPM of all the cores. The allocated size is first block aligned using $L = (bytenum + BLOCK\_SIZE - 1)/BLOCK\_SIZE * BLOCK\_SIZE$. Then, all the requested blocks are equally allocated from core 1 to core n, and the left blocks are allocated to the last core. Once that the SPM of all the cores can satisfy the request, the allocation is identified as an SPM allocation. The $flag$ = DISTRIBUTED_SPM; otherwise, if the SPM of one or more cores is not enough for allocation, the memory request is handled by both the SPM and the *malloc*() system call, in which situation the $flag$ = DISTRIBUTED_SPM_MEM.

Note that this interface will provide the programmer a logically flat address space to shield the physical discontinuity. From the perspective of the programmer, the distributed SPM allocation returns a continuous logic address space so that he/she will never be aware of the discontinuous memory spaces. In Solaris10, a virtual-physical address re-mapping is strictly limited by the OS. Thus, we indirectly implemented this

function by returning multiple pointers to the discontinuous memory spaces. However, an ideal solution is to implement the on-chip memory management support inside the OS, from which there is no technical obstacle to create the virtual-physical address mapping to hide the discontinuity at the physical address.

void SPM_free(void *addr). The free operation first compares the *addr* with the start address field in the allocation list node to search the matching allocation node. The allocation type is then known by resolving the flag. Once the SPM allocation is completely satisfied in the SPM, the free operation resets the related bitmap position according to the length field and deletes the node from the list; otherwise, a $free()$ system call is called to release the memory space allocated from the plain memory.

3.4 Optimization Based on Thread-Data Affinity

Multithreading has become the most popular technique to implement parallelism when programming the shared memory multicore architectures. The standardized C language programming interfaces that adhere to the IEEE POSIX 1003.1c standard are referred to as POSIX threads, or Pthreads. Pthread is a widely adopted multithreading standard for the Unix-like system. In a multithreading program, multiple threads may share the same array structure, each of which only accesses one segment of the array. In an extreme situation, *spm_normal_malloc*() allocates the whole array in the SPM of one core, and most of the threads access the data as remote references. This situation incurs a larger reference latency and a more serious competition for the interconnection, leading to an under-par performance. The remote accesses widely exist in the PGAS memory model due to the non-uniform access distance.

By considering the data-thread affinity, we optimize the SPMMLIB to support a distributed SPM allocation coupled with the system support to associate the thread with a certain core. Once the programmer allocates the SPM in a distributed manner, SPMMLIB returns the multiple addresses pointing to different cores' SPM. The returned pointers are then assigned as one of the parameters to the thread creation function, which creates and assigns the threads according to the related SPM core positions.

3.5 A Case Study

In this extreme example, the main function creates two individual threads that execute $\_func1()$ and $\_func2()$, respectively. The only activity of the two functions is to access the array, which is allocated in SPM by calling the *spm_distributed_malloc*() interface. Assuming that the program is executed in a dual-core machine. As expected, the array is allocated in the SPM of two cores with a discontinuous physical address space. To explore the data locality, two individual threads are created, and a reasonable enhancement is to associate the threads with the cores by the *processor_bind*() system call in Solaris, in order to make it closer to its local data.

Apparently, binding threads with the cores reduces the remote accesses overhead. However, it breaks the load balance mechanism when conducting the thread scheduling, which leads to an insufficient CPU utilization. A smarter scheme is to cancel the thread

binding after the SPM references, which can more reflect the different access pattern in different phases.

Listing 1.1. An example of using SPMMLIB in a multithreading program

```
#define ADDR int*
void* _func1(ADDR *testArray) {
    ...
    processor_bind(P_LWPID, P_MYID, 0, NULL);
    ADDR start_addr = testArray[0];
    for(i = 0; i < 2048; i++){
        start_addr[i] = i;
    }
    ...
}
void* _func2(ADDR *testArray){
    ...
    processor_bind(P_LWPID, P_MYID, 1, NULL));
    ADDR start_addr = testArray[1];
    for(i = 2048; i < 4096; i++){
        start_addr[i] = i;
    }
    ...
}
void main(int argc, char **argv) {
    ...
    ADDR *testArray;
    testArray =(ADDR *)spm_distributed_malloc(4096*sizeof(int));
    pthread_create(&t1, &attr1, _Func1, (ADDR *)testArray);
    pthread_create(&t2, &attr2, _Func2, (ADDR *)testArray);
    ...
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

4 Evaluation

4.1 Simulation Environment

We ran all experiments on the Wind River Simics simulator [16], which can support a full-system multiprocessor simulation. A quad-core processor is simulated, approximately simulating a tiled CMP consisting of multiple cores. The processor cores in the simulation configuration refer to the 512MHz UltraSPARC-III CPUs, with 4GB memory, and the compiler is gcc-3.4.6.

There are three on-chip memory architecture configurations in our performance comparison. (1) First is SPM-only, where every processor tile hosts 512KB SPM. The SPM is located at the same level with the L1 cache. The lack of tag logic enables a larger SPM in the same on-chip area compared with cache, and the SPM features a PGAS

Table 1. Description and the characteristics of the benchmarks

Benchmarks	Description	Heap size	SPM malloc
FFT	FFT algorithm	144KB	x, trans, umain, umain2
LU-CON	Contiguous LU algorithm	2060KB	proc_bytes, a , rhs, lc, y
LU-NCON	Non-contiguous LU	512KB	a, rhs, lc, y
RADIX	Radix sort algorithm	6726KB	key_partion,rank_partion, key[0], key[1]
OCEAN	Ocean movement simulation	13963KB	wrk2, wrk6, frcng
RADIOSITY	Radiosity algorithm	30237KB	task_buf, vis_struct
BLACKSCHOLES	Financial blackscholes model	256KB	data, prices, buffer, buffer2

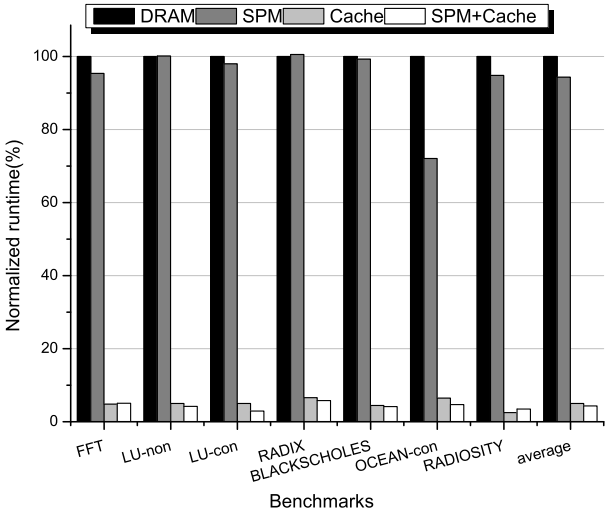


Fig. 4. Runtime comparison among DRAM, SPM, cache and SPM-cache

memory model. The processor core costs 1 cycle to access the local SPM while a remote access costs 15 cycles. (2) Second is cache-only, which contains a two-level cache hierarchy. The L1 cache of each core is composed of a 16KB data cache and a 16KB instruction cache, whereas the 2MB L2 cache is shared among all the processor cores. The L1 cache access latency is 1 cycle, and a L2 cache reference costs 18 cycles. (3) Third is SPM-cache, which shares the features of the two previous architectures. Both the SPM and the cache have the same parameters as with the previous settings. The off-chip DRAM access is configured to be 100 cycles in our experiment. Simics does not support the popular on-chip networks yet; thus, a simple bus interconnection is adopted in our simulation environment. However, note that the routing algorithm and the topology can affect the access latency of the SPM-based system. This paper only studies the data management in SPM, and thus we do not discuss the networking effect on the SPM accesses.

We selected six applications from the PARSEC [4] and SPLASH2 [26] benchmark suite. PARSEC has become more popular than SPLASH2 in recent years. However, our

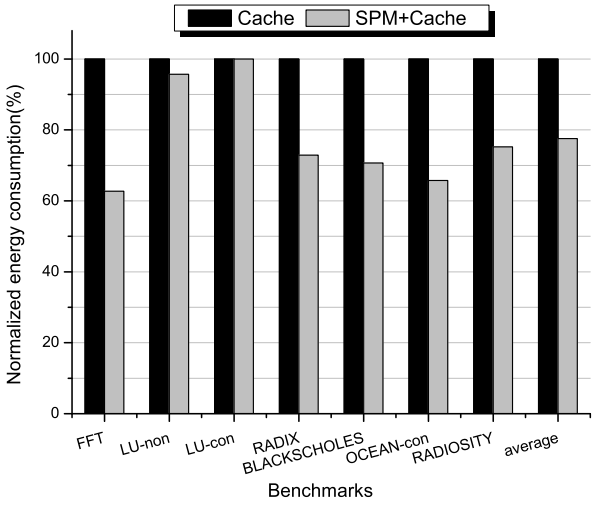


Fig. 5. Energy consumption comparison between cache and SPM-cache

SPMMLIB can only support the C programming language at present. The dynamic creation of the objects in an object-oriented language such as C++ is more complex. This is the reason why most of the PARSEC applications are abandoned in our experiment.

SPLASH2 is a multithreading benchmark suite that facilitates the study of the distributed shared memory space, which shares a target architecture similar to that of our study. By thoroughly analyzing the access pattern of the hot accessing data structures, we call the SPM management interfaces in SPMMLIB to replace the original malloc/free system calls. However, in a practical situation, the programmer is expected to know the access pattern of the code, which eases the use of our interfaces. We selected one benchmark from PARSEC and five benchmarks from SPLASH2. The details of the applications are listed in Table 1. The third column lists the heap data sizes in the benchmarks and the fourth column lists the heap data structures that are allocated to SPM through SPMMLIB.

4.2 Results

Fig. 4 shows the normalized runtime in a quad-core environment. We compare the runtime in DRAM, SPM, cache, and SPM-cache configurations. In the SPM and the SPM-cache configuration, the memory heap allocation system calls in the benchmarks are replaced with the *spm_normal_malloc*() in SPMMLIB. In this experiment, we set the thread number as the number of processors.

It is observed that cache and SPM-cache achieve much better performance than the DRAM and SPM configurations. The reason for this result is that most of the memory accesses are satisfied in cache, and only the heap data accesses are conducted in SPM. In most of the applications in the SPLASH-2 benchmark suit, heap data only

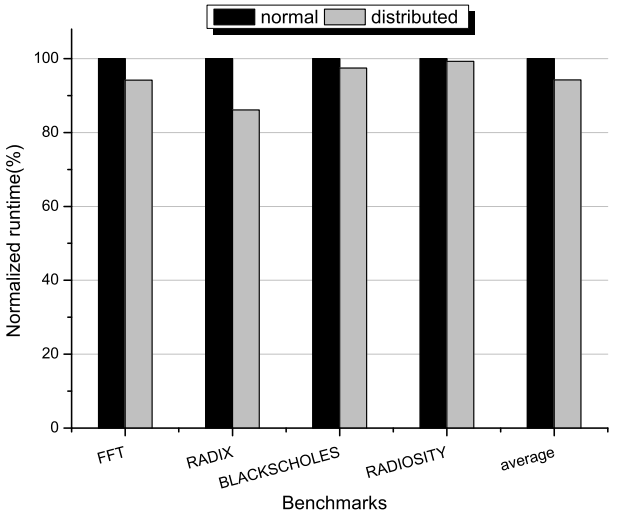


Fig. 6. Energy consumption comparison between cache and SPM-cache

occupies a small percentage of the total data items. Another reason is that the latency of SPM is in the same level with cache, thus, no very dramatic performance improvement is observed. However, this comparison is still meaningful, because the acceptable performance makes the SPM's innate advantages of energy and area promising and impressive.

Fig. 5 compares the energy consumption between SPM+cache and cache configurations. In the SPM+cache situation, the SPM is managed by SPMMLIB using *spm_normal_malloc*(). The energy consumption of a cache access and an SPM access are calculated using CACTIv5.3 [23], and the total energy is calculated using

$$Energy = Energy_{DRAM} + Energy_{cache} + Energy_{SPM} \quad (1)$$

From Fig. 5, we can observe that, in most benchmarks, the energy consumption of SPM+cache is much lower than the cache configuration. On average, the combination of cache plus SPM reduces the energy by about 22.4% compared with cache. The energy reduction is due to the SPM's innate power advantage over cache. By using SPMMLIB to allocate some heap data to SPM makes the SPM-located data uncacheable, which migrates considerable references from cache to SPM. Thus, the energy consumption is reduced. We believe that the energy profit gained by using SPM is an attractive reason to migrate SPM from embedded processors to the general-purpose desktop multicore processors. It is noted that the energy reduction is partially achieved by using a larger on-chip memory, compared with the pure cache reference configuration.

We tried to improve the performance by calling the *spm_distributed_malloc*() interface, in order to utilize the data-thread affinity. By thoroughly analyzing the benchmarks, the memory which was allocated in one tile is now being allocated in

different tiles. Then, multiple threads are created, each of which accesses one part of the heap data in its resided core. This evaluation is conducted in four benchmarks, including FFT, RADIX, BLACKSCHOLES, and RADIOSITY. Fig. 6 shows the performance comparison of the four selected benchmarks using *spm_normal_malloc*() and *spm_distributed_malloc*(), respectively. We can observe that an performance improvement is achieved in all the four benchmarks. On average, the distributed SPM allocation reduces the execution time by about 6% compared with the normal SPM allocation situation. Note that the distributed SPM allocation mode is more efficient for multithreading program, in which the multiple threads do not share the common data.

5 Related Work

In this section, we briefly review the previous literature on SPM management, which includes the SPM management strategies in single-core embedded systems and the methods for the multicore environment.

5.1 Traditional SPM Management

Traditional SPM research focuses on the space allocation for embedded applications. One of the most classic methods is to classify the SPM methods into static method and dynamic method, according to whether the SPM content can be overwritten during runtime. In the static methods, SPM space is not changed after the allocation is complete, whereas in the dynamic situation, SPM contents may be tuned in different execution phases. According to an accepted, the dynamic SPM management methods deliver better performance for the general-purpose applications, whose memory access pattern is more complex and is harder to predict. Egger [6] [10] proposed a horizontally partitioned memory architecture composed of SPM and a mini-cache, where the hot data are predicted using profiling knowledge and memory address redirection is addressed through the assistance of MMU. [20] and [9] made some contributions to the heap data management in SPM, which targets the different SPM architectures and requires the participation of the programmer. [27] proposed a new comparability graph coloring allocator that integrates data tiling and SPM allocation for arrays using tiling arrays on-demand to improve the utilization of the SPM, whereas in [18] and [17], the SPM allocation problem is formulated as an interval coloring problem. These methods are efficient for the general-purpose application's SPM management.

Traditional SPM management depends on the compiler or profiling knowledge to guide SPM space management. However, in some situations where the program's behavior is closely related to the outside input [5] and especially in the multitasking environment [11][22][12], predict the hot data at the compiling stage becomes difficult. The runtime SPM method [7][8] can reflect the changing access pattern at the cost of some hardware and runtime overhead.

5.2 SPM Management in CMP with LLM Features

Many studies discuss the local memory management in CELL BE processor, which is a typical model of the LLM architecture. There are one PPE and eight SPE, each of which

has a small on-chip local memory. SPE can only access the local memory, and the remote access of the other core's local memory is explicitly managed by the programmer through DMA. Data items in the local memory may have several copies, and there is no hardware protocol to maintain the coherence. [15] and [14] proposed a runtime system called COMIC, which provides the programmer with a vision of a globally shared memory. This framework assists the programmer to access data without worrying about the coherence problem. Another typical scheme is to manage the LLM as a software cache. It is a semi-automatic method to simulate hardware cache behaviors. The software actions incur a huge runtime overhead, and there is no very successful solution to it. Jaejin Lee et. al [21] proposed a software implementation called extended set-index cache (ESC), which is based on a four-way set-associative cache. Its tag lookup table has the same structure as that of the four-way set associative cache. However, it decouples the tag lookup table entries and the data blocks (i.e., the cache lines). The mapping between the tag entries and the data blocks is dynamically determined. This design combines the benefits of both set-associative cache and fully-associative cache. [2] focused on the heap management in CELL BE. A semi-automatic library for local memory heap allocation is developed to hide the complexity with a series of natrual programming interfaces. Their method is similar to that of our study, but there are some differences: (1) our method targets a class of architectures with the PGAS SPM feature, whereas their method manages the heap in LLM; (2) SPMMLIB takes the thread-data affinity into account to enhance the performance; and (3) their method is extended based on the CELL SDK, whereas our implementation is based on UltraSPARC architecture and supports a shared-distributed on-chip memory architecture.

5.3 SPM Management in CMP with PGAS Features

In the MPSoC architecture with PGAS SPM layout, [19] and [24] introduced two mechanisms to manage the on-chip memories. The study in [24] made contributions to the management of a single address space memory model in the CMP architecture, including both the static and dynamic data allocations in the software-managed on-chip memories. [19] introduced a programming framework based on OpenMP. The standard OpenMP APIs are extended to enable the programmer to express the need for optimized data partitioning and placement through annotations. A profile-based allocation compiler pass was also developed to assist the efficient SPM allocation at runtime. Our method distinguishes this study in two aspects. (1) The method in their literature is only an extension of OpenMP, where the annotation from the programmer is dealt with by the compiler. Our library delivers better portability because it is decoupled from the compiler and is developed in C. (2) The MPSoC in their method is based on RISC32 embedded cores, whereas the environment in our study is the Oracle/Sun UltraSPARC CMP using the Solaris OS. To the best of our knowledge, our evaluation is the first attempt to evaluate an SPM framework in the desktop multicore system.

6 Conclusion and Future Work

This paper introduces an SPM heap management framework, called SPMMLIB, for the future CMP with PGAS SPM feature. SPMMLIB provides some programming

interfaces to support the programmer to allocate the small on-chip memory when writing the code. The programmer only determines what data to be allocated, and SPMMLIB will take the responsibility of managing the data items in SPM. Thus, it eases the tedious and error-prone SPM management procedure with a simple and efficient dynamic library. SPMMLIB is independent from compiler and the profiling knowledge, which enables a good portability. We also enhance SPMMLIB using the data-thread affinity knowledge to reduce the remote accesses. Experimental results show that SPMMLIB can reduce the energy consumption by 22.4% compared with the cache while no notable degradation on performance.

We see several future directions for this study. Heap data is only a small part of the frequently accessed data in a program, it is possible to manage the stack and the static data using SPMMLIB. SPMMLIB supports only the C language, however, more and more programs are developed using an object-oriented language, and an object-oriented SPM management is also a promising topic. The creation of the object and the access pattern to an object are more complex.

References

1. Anzt, H., Hahn, T., Heuveline, V., Rocker, B.: GPU Accelerated Scientific Computing: Evaluation of the NVIDIA Fermi Architecture; Elementary Kernels and Linear Solvers. EMCL Preprint Series (2010)
2. Bai, K., Shrivastava, A.: Heap data management for limited local memory (llm) multi-core processors. In: Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES/ISSS 2010, pp. 317–326. ACM, New York (2010)
3. Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In: Proceedings of the Tenth International Symposium on Hardware/Software Codesign, CODES 2002, pp. 73–78. ACM, New York (2002)
4. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 72–81. ACM, New York (2008)
5. Cho, D., Pasricha, S., Issenin, I., Dutt, N.D., Ahn, M., Paek, Y.: Adaptive scratch pad memory management for dynamic behavior of multimedia applications. Trans. Comp.-Aided Des. Integ. Cir. Sys. 28, 554–567 (2009)
6. Cho, H., Egger, B., Lee, J., Shin, H.: Dynamic data scratchpad memory management for a memory subsystem with an mmu. SIGPLAN Not. 42(7), 195–206 (2007)
7. Deng, N., Ji, W., Li, J., Zuo, Q., Shi, F.: Core Working Set Based Scratchpad Memory Management. IEICE Transactions on Information and Systems 94(2), 274–285 (2011)
8. Deng, N., Ji, W., Li, J., Shi, F., Wang, Y.: A novel adaptive scratchpad memory management strategy. In: Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009, pp. 236–241. IEEE Computer Society, Washington, DC, USA (2009)
9. Dominguez, A.: Heap data allocation to scratch-pad memory in embedded systems. Technical report, University of Maryland at College Park, College Park, MD, USA, AAI3260306 (2007)
10. Egger, B., Lee, J., Shin, H.: Scratchpad memory management for portable systems with a memory management unit. In: EMSOFT 2006: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software, pp. 321–330. ACM, New York (2006)

11. Egger, B., Lee, J., Shin, H.: Scratchpad memory management in a multitasking environment. In: EMSOFT, pp. 265–274 (2008)
12. Ji, W., Deng, N., Shi, F., Zuo, Q., Li, J.: Dynamic and adaptive spm management for a multi-task environment. J. Syst. Archit. 57, 181–192 (2011)
13. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev. 49, 589–604 (2005)
14. Lee, J., Lee, J., Seo, S., Kim, J., Kim, S., Sura, Z.: COMIC++: A software SVM system for heterogeneous multicore accelerator clusters. In: 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA 2010), pp. 1–12. IEEE, Los Alamitos (2010)
15. Lee, J., Seo, S., Kim, C., Kim, J., Chun, P., Sura, Z., Kim, J., Han, S.: Comic: a coherent shared memory interface for cell be. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, pp. 303–314. ACM, New York (2008)
16. Leupers, R.: Processor and system-on-chip simulation. Springer, Heidelberg (2010)
17. Li, L., Feng, H., Xue, J.: Compiler-directed scratchpad memory management via graph coloring. ACM Trans. Archit. Code Optim. 6, 9:1–9:17 (2009)
18. Li, L., Xue, J., Knoop, J.: Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs. ACM Trans. Embed. Comput. Syst. 10, 28:1–28:42 (2011)
19. Marongiu, A., Benini, L.: An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs. IEEE Transactions on Computers (2010)
20. McIlroy, R., Dickman, P., Sventek, J.: Efficient dynamic heap allocation of scratch-pad memory. In: ISMM 2008: Proceedings of the 7th International Symposium on Memory Management, pp. 31–40. ACM, New York (2008)
21. Seo, S., Lee, J., Sura, Z.: Design and implementation of software-managed caches for multicores with local memory. In: IEEE 15th International Symposium on High Performance Computer Architecture, HPCA 2009, pp. 55–66. IEEE, Los Alamitos (2009)
22. Takase, H., Tomiyama, H., Takada, H.: Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2010, pp. 1124–1129. European Design and Automation Association, Leuven (2010)
23. Thoziyoor, S., Muralimanohar, N., Ahn, J.H., Jouppi, N.P.: CACTI 5.3, HP Laboratories Palo Alto (2009)
24. Villavieja, C., Gelado, I., Ramirez, A., Navarro, N.: Memory Management on Chip-MultiProcessors with on-chip Memories. In: Proc. Workshop on the Interaction between Operating Systems and Computer Architecture (2008)
25. Walleij, L.: Arm tcm (tightly-coupled memory) support v3 (2009)
26. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 24–36. ACM, New York (1995)
27. Yang, X., Wang, L., Xue, J., Tang, T., Ren, X., Ye, S.: Improving scratchpad allocation with demand-driven data tiling. In: Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2010, pp. 127–136. ACM, New York (2010)

Parallel Binomial Valuation of American Options with Proportional Transaction Costs

Nan Zhang[1], Alet Roux[2], and Tomasz Zastawniak[2]

[1] Department of Computer Science and Software Engineering,
Xi'an Jiaotong-Liverpool University, China
nan.zhang@xjtlu.edu.cn
[2] Department of Mathematics, University of York, UK
{alet.roux,tomasz.zastawniak}@york.ac.uk

Abstract. We present a multi-threaded parallel algorithm that computes the ask and bid prices of American options with the asset transaction costs being taken into consideration. The parallel algorithm is based on the recombining binomial tree model, and is designed for modern shared-memory multi-core processors. Although parallel pricing algorithms for American options have been well studied, the cases with transaction costs have not been addressed. The parallel algorithm was implemented via POSIX Threads, and was tested. The results demonstrated that the approach was efficient and light-weighted. Reasonable speedups were gained on problems of small sizes.

Keywords: Parallel computing; multi-core processing; option pricing; binomial process; transaction costs.

1 Introduction

An American call (put) option is a financial derivative contract which gives the option holder the right but not the obligation to buy (sell) one unit of a certain asset (stock) for the exercise price K at any time until a future expiration date T. Option pricing is the problem of computing the price of an option. It is crucial to many financial practices. Since the classic work on this topic by Black, Scholes and Merton [2,5], many new developments have been introduced. In this paper, we present a parallel algorithm and its multi-threaded implementation that computes the ask and bid prices of American options when proportional transaction costs in the underlying asset trading are taken into consideration. Previous work on parallel valuation of European and/or American options can be found, e.g., in [3,6,4,10], but none of these took into consideration the issue of transaction costs. The algorithm that we designed parallelises the sequential pricing algorithms proposed by Roux and Zastawniak [9]. The parallelisation was based on the binomial tree model and was implemented via POSIX Threads. The implementation was tested on a machine equipped with 8 processors. When the number N of time steps was 1200 the speedup of the parallel against the sequential was about 5. Comparing this to the results obtained by the previous

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 88–97, 2011.

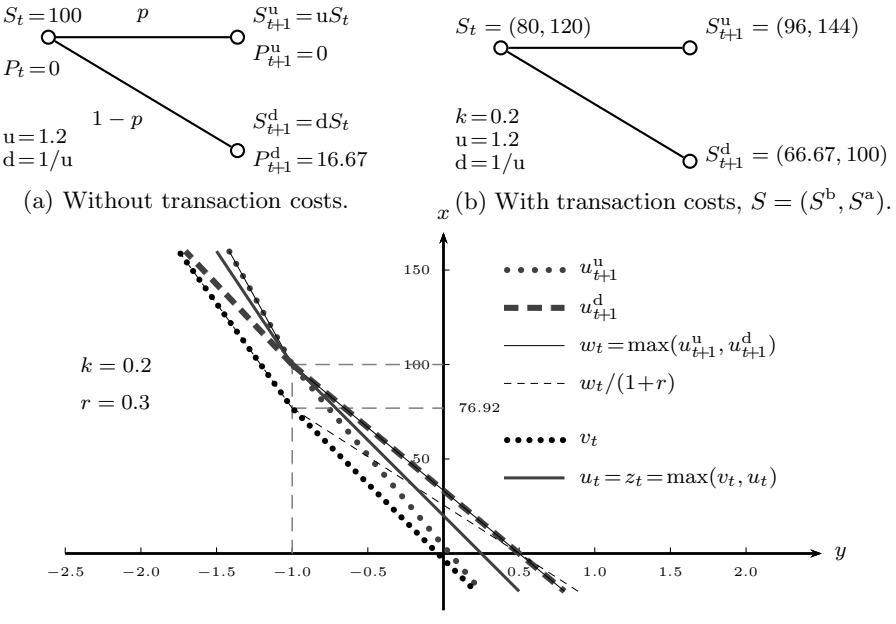
papers [3,6,4] where reasonable speedups were achieved only after N grew to 8192, this multi-threaded approach reduced the cost of parallelisation and gained speedups on problems of much smaller sizes.

The contributions of this work are twofold. First, the parallel algorithm takes into account the issue of transaction costs, whereas previous work for the same problem did not. Second, an improved generic parallel strategy for partitioning a binomial tree is developed. This partition scheme re-calculates the load of each processor before each new round of computation starts, thus re-balancing the workload of the processors dynamically. However, the partition scheme found in the previous works [3,6,4,10] fixed the assignment of blocks to processors at the starting of the computation. The partition scheme is generic because its applicability is not confined by the values of the model parameters.

Organisation of the rest of this paper. Related work is reviewed in Section 2. The sequential pricing algorithms are briefly explained in Section 3. The parallel algorithm and its analysis are presented in Section 4. Experimental results are reported in Section 5. Conclusions are drawn in Section 6, which also contains a discussion on future work.

2 Related Work

The algorithms proposed by Roux and Zastawniak [9] are preferable to the earlier works [8,7] for pricing American options with transaction costs, because the applicability of the algorithms is not confined by the values of certain market and model parameters, or by methods of settlement (cash or physical delivery of the underlying asset). Like the previous works [3,6,4,10], our parallel algorithm is designed for recombining binomial trees so that the runtime of the algorithm is polynomial in N (the number of time steps), rather than exponential. Gerbessiotis [3] presented a parallel American option pricing algorithm on binomial trees, where a tree was partitioned into $b \times b$ blocks. The blocks were assigned to distinct processors. The whole process was divided into a number of rounds, but no load re-balancing was found after each round of computation. The algorithm was implemented via the BSP (Bulk Synchronous Parallel) model [1], and was tested on a 16-node PC cluster. The speedup against the serial computation was 2.97 (Table 1 in [3]) using all the 16 nodes when $N = 8192$ and $b = 64$. Pend and Gong et al. [6] presented their binomial-tree-based parallel option pricing algorithm, which assumed N and p (number of processors) to be a power of two. Again, load re-balancing was not found in the algorithm. Their testing was done on a 16-node PC cluster. A speedup of 3.33 (Table 1 in [6]) was reported when $N = 8192$ and $p = 16$. Zubair and Mukkamala [10] presented a cache-efficient parallel binomial option pricing algorithm, which was tested on 8 Sun UltraSPARC III processors and 4.96 times (Table 4 in [10]) speedup was found when $p = 8$ and $N + 1 = 8192$. Compared with the above approaches, the multi-threaded solution that we propose makes no assumptions about the number of time steps or the block size. It does dynamic load re-balancing and is much more light-weighted.



(a) Without transaction costs.

(b) With transaction costs, $S = (S^{\mathrm{b}}, S^{\mathrm{a}})$.

(c) The seller's hedging strategy.

Fig. 1. One-step binomial pricing with/without transaction costs

3 The Sequential Pricing Algorithms

We first briefly go through the idea of pricing American options when transaction costs are not considered. We use the one-step binomial process example in Fig. 1(a), where at time t the price of the underlying stock of an American put option is S_t. The exercise price of the put is $K = 100$. After the one time step, the price of the stock can either be $\mathrm{u}S_t$ or $\mathrm{u}^{-1}S_t$. The risk-neutral probability for an up-move is p, and for the down-move is $1-p$. We assume the interest rate over the one step time period is r, that is, 1 unit of cash bond at time t will grow to $1+r$ units at time $t+1$. The payoff P_t of such a put option at some time t is $P_t = \max(K - S_t, 0)$. So the value π_t of the option at node S_t is the maximum of its discounted expected payoff $(1+r)^{-1}\mathbb{E}(\pi_{t+1}|S_t) = (1+r)^{-1}(pP^{\mathrm{u}}_{t+1} + (1-p)P^{\mathrm{d}}_{t+1})$ at time t and its immediate payoff P_t if the option is exercised at t, that is $\pi_t = \max(P_t, \mathbb{E}(\pi_{t+1}|S_t)/(1+r))$. To compute π_0 on a binomial tree of multiple levels, we start from the leaf nodes and go all the way back to the root node to obtain the option's price at time 0.

Proportional transaction costs are manifested in asset (stock) tradings as the ask-bid spread, that is, a unit of a stock can be bought at time t by the ask price $S^{\mathrm{a}}_t = (1+k)S_t$, and sold for the bid price $S^{\mathrm{b}}_t = (1-k)S_t$, where k is the transaction cost rate. Under such conditions the price of an American option at any time t is no longer unique, but is confined within an interval if no arbitrage opportunity is to be found. The upper limit of this interval is the ask price π^{a}_t of

the option, and the lower is the bid price π_t^{b}. The ask price is the price at which the option can be bought for sure. It is also the minimum amount of wealth that the seller of the option needs to cover his/her positions in all circumstances, that is, to deliver to the buyer the obligated payoff portfolio without having to inject extra wealth. The bid price is the price at which the option can be sold for sure. It is also the maximum amount of wealth that the buyer can borrow against the right to exercise the option.

Let (ξ, ζ) be the payoff process of an American option, that is, at each time $t = 0, 1, 2, \ldots, T$, if the holder exercises the option the seller must deliver to the holder a portfolio consisting of ξ_t cash and ζ_t units of the asset (stock). For the above American put option this is $(K, -1)$ at all times. To hedge his/her position the seller should hold a portfolio consisting of cash bonds and the underlying stock, and we use (x_t, y_t) to denote his/her holdings at time t. We define the seller's expense function u_t at time t to be $u_t(y) = \xi_t + (y-\zeta_t)^- S_t^{\mathrm{a}} - (y-\zeta_t)^+ S_t^{\mathrm{b}}$, where $(y-\zeta_t)^- = -\min(y-\zeta_t, 0)$ and $(y-\zeta_t)^+ = \max(y-\zeta_t, 0)$. This is a function of the seller's stock holding y_t at time t. It defines the minimum amount of cash that the seller needs at t to fulfil his/her obligation if the option is exercised at t. So if the seller wishes to form a self-financing strategy to cover his/her position at t, his/her holdings (x_t, y_t) must belong to the epigraph of u_t, that is $(x_t, y_t) \in \operatorname{epi} u_t$. (The epigraph of u_t is the set of points which lie above u_t in the $y - x$ plane, $\operatorname{epi} u_t = \{(x_t, y_t) \in \mathbb{R}^2 \mid x_t \geq u_t(y_t)\}$.)

Now using the same American put ($K = 100$) example and the one-time step binomial process (Fig. 1(b)) we explain how the option ask price π_t^{a} at time t is computed. We start from the two nodes at time $t+1$. The seller's expense function at the up-move node is $u_{t+1}^{\mathrm{u}}(y) = \xi_{t+1} + (y-\zeta_{t+1})^- S_{t+1}^{\mathrm{a}} - (y-\zeta_{t+1})^+ S_{t+1}^{\mathrm{b}} = 100 + 144(y+1)^- - 96(y+1)^+$. This is a piecewise linear function because, when $y < -1$, $u_{t+1}^{\mathrm{u}} = -144y - 44$, and when $y \geq -1$, $u_{t+1}^{\mathrm{u}} = -96y + 4$, as they are shown in Fig. 1(c). For the down-move node $u_{t+1}^{\mathrm{d}}(y) = 100 + 100(y+1)^- - 66.67(y+1)^+$. At time t, because the seller must prepare for the worst case, we calculate the maximum of u_{t+1}^{u} and u_{t+1}^{d}, and the result is denoted by w_t. Now because the interest rate over the one time step is r, so x_t units of cash will grow to $x_t(1+r)$ at time $t+1$, and therefore, the function w_t must be discounted by $(1+r)$. Now the slopes of this discounted function $w_t/(1+r)$ must be restricted within the interval $[-S_t^{\mathrm{a}}, -S_t^{\mathrm{b}}]$, which is $[-120, -80]$ in this example. This restricted function is then denoted by v_t, and this is the discounted expected expense function at time t supposing the option is exercised at time $t+1$. Now what if the option is exercised at time t? The expense function u_t at t is $u_t = 100 + 120(y+1)^- - 80(y+1)^+$. Again, the seller must prepare for the worst, so the maximum function z_t for the worst case at t is $z_t = \max(u_t, v_t)$. The option ask price π_t^{a} for this example is then $\pi_t^{\mathrm{a}} = z_t(0)$. When the above computation is carried out on a binomial tree representing N time steps, we start from the leaf nodes and work backwards to the root node at time 0. So the option ask price $\pi_0^{\mathrm{a}} = z_0(0)$, because it is the minimum amount of wealth that enables the seller to hedge his/her positions.

For the buyer's case, the expense function at time t is $u_t(y) = -\xi_t + (y + \zeta_t)^- S_t^{\mathrm{a}} - (y+\zeta_t)^+ S_t^{\mathrm{b}}$, because it is he/she who will receive the portfolio (ξ_t, ζ_t).

Whenever z_t is computed the minimum operation is used instead of the maximum, because the buyer would wish to minimise his/her expenses when he/she chooses to exercise the option. When the computation is on a binomial tree of N steps, at the end of the process, the bid price $\pi_0^{\mathrm{b}} = -z_0(0)$ because it is the maximum amount of wealth that the buyer can borrow to protect his/her positions. Full details about these can be found in Algorithm 3.1 and 3.5 in [9].

4 The Parallel Algorithm

For an American option whose payoff process and expiration time are (ξ, ζ) and T, respectively, let N be the number of time intervals that discretise the time period from 0 to T, σ the volatility of the underlying stock, R the continuously compounded annual interest rate and $k \in [0, 1)$ the transaction cost rate. Under such conditions the binomial tree that models the price changing of the stock will have $N+1$ levels, corresponding to the time steps $t = 0, 1, 2, \ldots, N$. The up-move factor u, down-move factor d and interest rate r over one time step are $\mathrm{u} = \exp(\sigma\sqrt{T/N})$, $\mathrm{d} = \exp(-\sigma\sqrt{T/N}) = \mathrm{u}^{-1}$, and $r = \exp(RT/N)$, respectively. However the pricing algorithms [9] actually add an extra time instant $t = N + 1$ to the model and set the option payoff as $(0, 0)$ at all the $N + 2$ nodes in that level. The purpose of adding this extra time step is to model the possibility that under certain circumstances the option may never be exercised by the option holder.

Now assume we have p distinct processors, denoted by $p_0, p_1, \ldots, p_{p-1}$. Because the computation of the u, w, v, z functions on different nodes can be performed independently in parallel, we can partition the whole tree into blocks of nodes and assign these blocks to the distinct processors. The parallel algorithm, like its sequential counterpart, starts off at the leaf nodes where $t = N + 1$ and works backwards to the root. The whole process is finished by p threads, with each thread being bound explicitly onto a distinct processor. We thus use p_i to denote the thread that has been bound onto processor $p_i, i \in [0, p-1]$. The whole computation is divided into a number of rounds, where each thread, in parallel, processes the part of the tree that is assigned to it. In general, if the base level of the current round is $t = n, n \in [1, N + 1]$, the total number of nodes in the level is $n+1$. These $n+1$ nodes will be divided equally among the p threads. So all the threads $p_i, i = 0, 1, \ldots, p-2$, get $\lfloor (n + 1)/p \rfloor$ nodes, but the last thread p_{p-1} gets $(n + 1) - \lfloor (n + 1)(p - 1)/p \rfloor$ nodes. We use L to denote the maximum number of levels that are processed towards the root in a round. However, the number D of levels that are actually processed in a round is jointly determined by L and the number of nodes that each thread gets, because this number D cannot exceed $\lfloor (n + 1)/p \rfloor - 1$. So we have $D = \min(L, \lfloor (n + 1)/p \rfloor - 1)$. So in the round where $t = n$ all the threads will be assigned a block of $\lfloor (n+1)/p \rfloor \times D$ nodes, except the last thread p_{p-1}. For a thread $p_i, i \in [0, p-2]$, we further divide its $\lfloor (n+1)/p \rfloor \times D$ nodes into region A and region B such that the computation performed at nodes in region A do not have to wait for the results from thread p_{i+1}, but the computation at nodes in region B do have to wait. Note that the

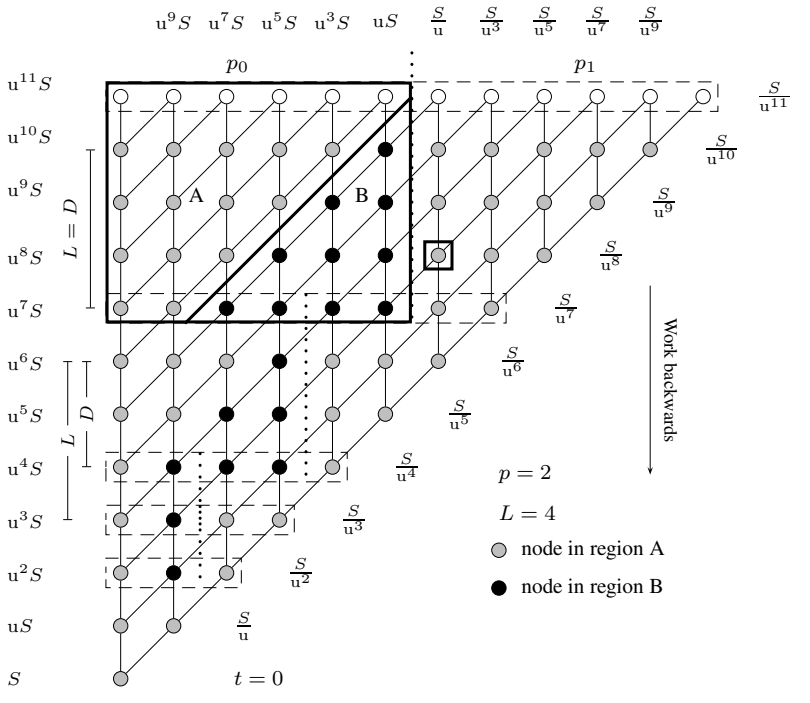


Fig. 2. Parallel processing on the binomial tree by two working threads

last thread p_{p-1} does not have B nodes in all rounds of the process. The parallel algorithm re-balances the workload among the threads after each round. If the current base level is n, the next base level will be $n - D$ containing $n - D + 1$ nodes, which will be divided equally among the threads. The parallel algorithm ensures that each thread will get minimally two nodes to process in all the rounds, which means that the minimum possible value that D can get is 1. If at some level of the tree the number of nodes is less than $2p$ then the number of processors used will be decreased until this no-less-than-two-node condition is satisfied. A partition based on the above explanation is shown in Fig. 2 where we have $N = 10$, $p = 2$ and $L = 4$.

To save the intermediate z functions generated during the process, instead of generating the whole tree, the parallel algorithm maintains two buffers, each with $(L + 1)$ rows $\times$ $(N + 2)$ columns. One of these two buffers is for computing the ask price, and the other the bid price. Because the whole process is divided into rounds the threads have to be synchronised. For threads $p_i, i = 0, 1, \ldots, p-1$, in general, in a round they have to work D levels down the tree towards the root. As soon as thread $p_i, i \in [1, p-1]$ finishes the leftmost node (the single node enclosed by the bold frame in Fig. 2) at level $B - D + 1$ (B being the base level of the tree in the current round) in its A region, it will send a signal to thread p_{i-1}, so that after thread p_{i-1} finishes the nodes in its A region it can continue on the ones in the B region. Once thread $p_i, i \in [1, p-1]$ finishes processing all

Algorithm 1. Computation by thread $p_i, i \in [0, p-1]$

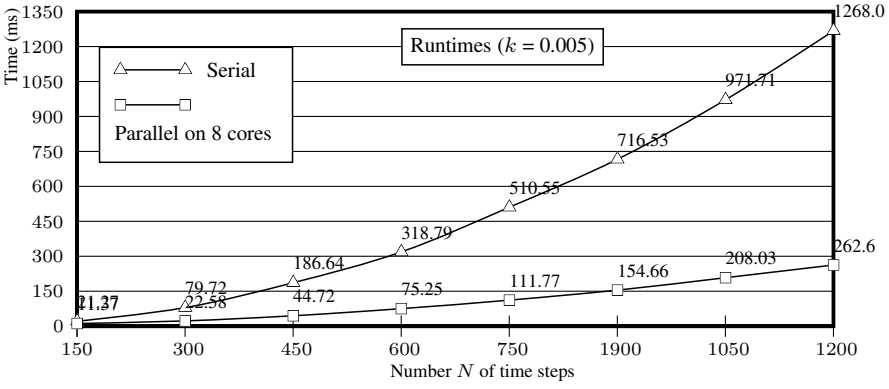
Input: Up-move factor u, interest rate r, number p of processors, number N of time steps, stock price S_0 at root node, transaction cost rate k.

Output: The functions u, w, v and z at each node.

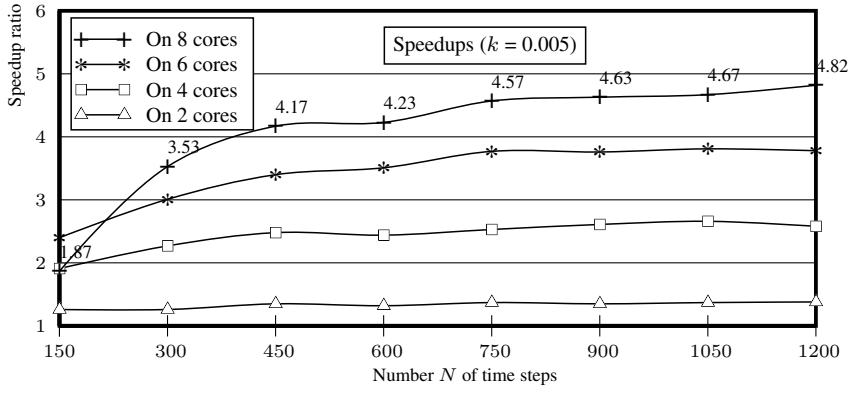
```
1  begin
       // Initialisation at nodes in level t = N + 1.
2      n ← N + 1, s ← i × ⌊(n + 1)/p⌋;
3      e ← (i + 1) × ⌊(n + 1)/p⌋ ∀ i ≠ p − 1, or e ← n + 1 when i = p − 1;
4      Set z_{N+1} = u_{N+1} for the seller and the buyer with payoff (0,0) at nodes in
       level N + 1 within range [s, e − 1] ;

       // Start to work backwards down to the root.
5      for B ← N + 1; B > 0 and i < p; do
6          Compute the z functions at nodes in region A from level B − 1 to
           B − D within [s, e − 1];
7          if i > 0 then
8              Send signal G_i to thread p_{i−1} once the z function has been
               computed at the leftmost node at level B − D + 1;
9          if i + 1 < p then
10             Wait until receives signal G_{i+1};
11             Compute the z functions at nodes in region B from level B − 1 to
               B − D within [s, e − 1];
12             Send signal C_{i+1} to thread p_{i+1} for it to start the copy back;
13         if i > 0 then
14             Wait until receives signal C_i;
15         Copy back the computed z functions;
16         Wait until all threads reach this point;
17         B ← B − D;
18         if B > 0 then
19             n ← B + 1;
20             while n < 2 × p do
21                 p ← max(p/2, 1);
22             s ← i × ⌊(n + 1)/p⌋;
23             e ← (i + 1) × ⌊(n + 1)/p⌋ ∀ i ≠ p − 1, or e ← n + 1 when i = p − 1;
24 end
```

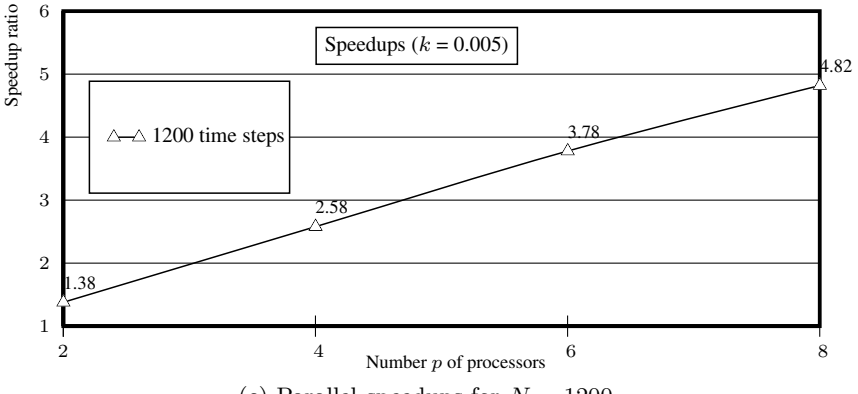
its nodes in A and B regions, it has to wait for thread p_{i-1} to finish processing its B region, then can thread p_i copy the z functions stored in the Dth rows of the two buffers back to the first rows for next round of computation. But for thread p_0 it can make the copy back once it finishes processing its A and B regions without having to wait. Once all the threads have finished processing in a round the parameters will be updated to make preparations for the next round. Algorithm 1 shows the steps carried out by thread $p_i, i \in [0, p-1]$ and the synchronisation scheme which were achieved by semaphores and conditional variables.



(a) Runtimes of the sequential and the parallel implementation on 8 processors.



(b) Parallel speedups against the sequential.



(c) Parallel speedups for $N = 1200$.

Fig. 3. Experimental results

The sequential pricing algorithms have polynomial runtime $T_{\mathrm{S}} = O(N^k)$ for some $k \geq 2$. Although the number of nodes in a binomial tree is quadratic in N, the maximum, minimum and slope restriction operations may require slightly

more time to finish as the computation backs towards the root because the piecewise linear functions u, w, v and z may get more pieces at nodes closer to the root. Roughly speaking, the parallel algorithm, because it divides the computation equally among p processors, its parallel runtime is $T_{\mathrm{P}} = O(N^k/p)$. So the speedup $S = T_{\mathrm{S}}/T_{\mathrm{P}} = O(p)$, proportional to the number p of processors used.

5 Experimental Results

The parallel pricing algorithm that computes the π_0^{a} and π_0^{b} was implemented in C/C++, and was tested on a machine with dual sockets × quad-core Intel Xeon E5405 running at 2.0GHz – 8 processors in total. The source code was compiled by Intel C/C++ compiler icpc 12.0 for Linux. The testing machine was running Ubuntu Linux 10.10 64-bit version. The POSIX thread library used was NPTL (native POSIX thread library) 2.12.1.

To verify the correctness of the parallel algorithm we computed the ask and bid prices for the same American put option as it is described in Example 5.1 in [9] where $T = 0.25$, $\sigma = 0.2$, $R = 0.1$, $S_0 = 100$, $K = 100$, N varied from 20 to 1000 and k from 0 to 0.02. In all the cases the parallel implementation produced exactly the same figures as reported in Table 1 in [9].

To test the performance of the parallel implementation against an optimised sequential program another group of tests were made where k was fixed to 0.005, but N varied from 150 to 1200, and p from 2 to 8. The results are reported in Fig. 3. All the times were wall-clock times measured in milliseconds (ms).

The experimental results showed that performance of the parallel algorithms was in-line with the theoretical analysis. The parallel speedup was proportional to p, the number of processors (Fig. 3(c)). The multi-threaded approach was light-weighted in that reasonable speedups were gained for problems of small sizes, e.g., when $N = 300$ (Fig. 3(b)). The parallel efficiency when $N = 1200$ and $p = 8$ were above 60%, better than some of the previous approaches based on message passing schemes without taking transaction costs into consideration.

6 Conclusion

We have presented a parallel algorithm that computes the ask and bid prices of American options under proportional transaction costs, and a multi-threaded implementation of the algorithm. Using p processors, the algorithm partitions a recombining binomial tree into blocks and assigns these blocks to distinct processors for parallel processing. The whole process, starting from the leaf nodes and working backwards to the root, is divided into a number of rounds. After each round of computation the workload of each processor (thread) is re-calculated and thus re-balanced. The partition method and the associated synchronisation scheme are generic in the sense that their applicability is not confined by the values of the parameters N (number of levels of the tree), L (maximum number of levels processed in a round) or p (number of processors). The parallel algorithm

has theoretical speedup $S = O(p)$ and is therefore cost-optimal because the product of the parallel runtime T_{P} and p is $pT_{\mathrm{P}} = O(p) \times O(N^k/p) = O(N^k)$ for some $k \geq 2$, which has the same asymptotic growth rate as the serial runtime T_{S}.

The implementation was tested for its correctness and performance. The results demonstrated reasonable speedups against an optimised sequential computation even on problems of small sizes. The speedups were in-line with the asymptotic analysis, and showed that because no inter-computer communication is involved in the multi-threaded approach, the overhead of the parallelisation was much reduced comparing to some previous approaches based on message-passing interfaces.

In future, we can extend this work by adapting this parallel algorithm to general purpose graphics units, as well as conducting tests on more up-to-date parallel architectures of a larger scale.

References

1. Bisseling, R.H.: Parallel Scientific Computation: A Structured Approach using BSP and MPI. Oxford University Press, Oxford (2004)
2. Black, F., Scholes, M.: The Pricing of Options and Corporate Liabilities. The Journal of Political Economy 81(3), 637–659 (1973)
3. Gerbessiotis, A.V.: Architecture Independent Parallel Binomial Tree Option Price Valuations. Parallel Computing 30, 301–316 (2004)
4. Gerbessiotis, A.V.: Parallel Option Price Valuations with the Explicit Finite Difference Method. International Journal of Parallel Programming 38, 159–182 (2010)
5. Merton, R.: Theory of Rational Option Pricing. Bell Journal of Economics and Management Science 4, 141–183 (1973)
6. Peng, Y., Gong, B., Liu, H., Zhang, Y.: Parallel Computing for Option Pricing Based on the Backward Stochastic Differential Equation. In: Zhang, W., Chen, Z., Douglas, C.C., Tong, W. (eds.) HPCA 2009. LNCS, vol. 5938, pp. 325–330. Springer, Heidelberg (2010)
7. Perrakis, S., Lefoll, J.: Option Pricing and Replication with Transaction Costs and Dividends. Journal of Economic Dynamics & Control 24, 1527–1561 (2000)
8. Perrakis, S., Lefoll, J.: The American Put under Transactions Costs. Journal of Economic Dynamics & Control 28, 915–935 (2004)
9. Roux, A., Zastawniak, T.: American Options under Proportional Transaction Costs: Pricing, Hedging and Stopping Algorithms for Long and Short Positions. Acta Applicandae Mathematicae 28, 199–228 (2009)
10. Zubair, M., Mukkamala, R.: High Performance Implementation of Binomial Option Pricing. In: Gervasi, O., Murgante, B., Laganà, A., Taniar, D., Mun, Y., Gavrilova, M.L. (eds.) ICCSA 2008, Part I. LNCS, vol. 5072, pp. 852–866. Springer, Heidelberg (2008)

A Parallel Analysis on Scale Invariant Feature Transform (SIFT) Algorithm

Donglei Yang, Lili Liu, Feiwen Zhu, and Weihua Zhang

Parallel Processing Institute, Fudan University
{ydl,llliu,zhufeiwen,zhangweihua}@fudan.edu.cn

Abstract. With explosive growth of multimedia data on internet, the effective information retrieval from a large scale of multimedia data becomes more and more important. To retrieve these multimedia data automatically, some features in them must be extracted. Hence, image feature extraction algorithms have been a fundamental component of multimedia retrieval. Among these algorithms, Scale Invariant Feature Transform (SIFT) has been proven to be one of the most robust image feature extraction algorithm. However, SIFT algorithm is not only data intensive but also computation intensive. It takes about four seconds to process an image or a video frame on a general-purpose CPU, which is far from real-time processing requirement. Therefore, accelerating SIFT algorithm is urgently needed. As multi-core CPU becomes more and more popular in recent years, it is natural to employ computing power of multi-core CPU to accelerate SIFT. How to parallelize SIFT to take full use of multi-core capabilities becomes one of the core issues. This paper analyzes available parallelism in SIFT and implements various parallel SIFT algorithms to evaluate which is the most suitable for multi-core system. The final result shows that our parallel SIFT achieves a speedup of 10.46X on 16-core machine.

1 Introduction

With rapid development of Internet and cloud computing, our society has turned into a data-centric world. A huge amount of data is processed every minute. In terms of the prediction of CISCO Inc [1], until 2014, the data quantity generated every month will reach about 0.6 million PB. Among them, multimedia data, such as images and videos has become one of the most common data types. With the scale of multimedia data increasing, it is vitally important to efficiently extract useful information from a huge amount of multimedia data. As a fundamental component of multimedia retrieval applications, image feature extraction algorithms have become one of research hotspots [12,13,14]. Generally, image extraction algorithms can be divided into two classes: global feature-based algorithms and local feature-based ones. Global feature-based algorithms usually use a single feature, such as color or texture, to represent an image. It is fast for global feature-based algorithms to extract the features and do the matching. However, the precision rate for these algorithms is low (more than 30% error

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 98–111, 2011.

rate) [17]. Moreover, these global feature-based algorithms cannot retrieve an image when there are different transformations for it, such as resizing and cropping [18]. In contrast, local feature-based algorithms have high precision and are insensitive to different transformations. As a result, they heve been widely used in different applications [2,3].

The most widely used local feature-based algorithms are SIFT (Scale Invariant Feature Transform) [4,5] and SURF (Speeded Up Robust Features) [8]. They extract hundreds or thousands of feature points to represent one image and are invariant to rotation, scaling, brightness contrast and view point changing. Although SURF is faster than SIFT, SIFT has been proven to be the most robust image feature extraction algorithm [6], and has been more widely used than SURF.[1]

However, due to the complexity of the algorithms, SIFT is not only data intensive but also computation intensive. It takes about four seconds to process an image or a video frame on a general-purpose CPU. Therefore, with the explosive growth of multimedia data on internet, accelerating SIFT algorithm is urgently needed.

As multi-core CPU becomes more and more popular in recent years, it is natural to employ computing power of multi-core CPU to accelerate SIFT. The core issues of accelerating SIFT by multi-core system are whether we can parallelize SIFT algorithm and how to take full use of multi-core computing power to gain a good scalability. In this paper, we first analyze the parallel characteristics of SIFT, and then implement the various parallel designs of SIFT to evaluate which is the most suitable architectures for multi-core system. The result shows that our parallel implementation achieves a speedup of 10.46X on 16-core machine.

The paper is organized as follows. Section 2 introduce SIFT algorithm in brief and then time analysis and parallel characteristics analysis are given. In section 3, we implement various parallel versions based on parallel characteristics analysis including scale level parallelism and block level parallelism. In section 4, the evaluation shows the performance overview of all implementations. In Section 5, the related work is discussed. Section 6 concludes the paper and discusses the future work.

2 Computing SIFT

Scale Invariant Feature Transform (SIFT) is a computer vision algorithm to extract local features in images. After it is published by David Lowe [4,5] in 1999, it has been widely used in many applications, such as object recognition, gesture recognition and match moving. In this section, we first introduces SIFT algorithm in brief. And then time analysis and parallel characteristics analysis are proposed, which helps to the future parallelism design and implementation.

[1] According to the return results of google scholar, 9666 papers are related with SIFT, while 1201 papers are related with SURF.

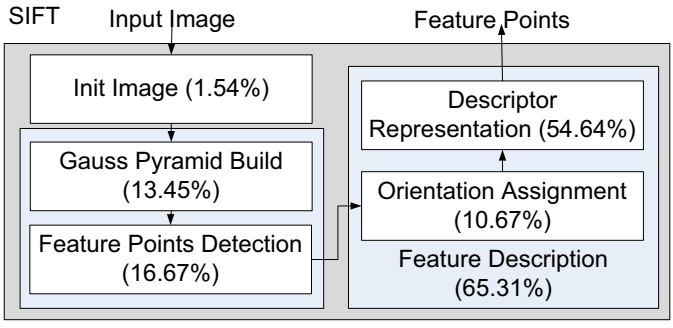


Fig. 1. The workflow of the SIFT algorithm

2.1 SIFT Overview

As shown in Fig. 1, SIFT consists of three major stages: 1) Build Scale-Space; 2)Feature Points Detection; 3)Description Stage. The workflow of SIFT is described as follows:

Gauss Pyramid Build. To achieve invariant to scale, gauss pyramid is constructed. Gauss pyramid contains several octaves and each octave consists of several intervals which are computed from the original image through a repeatedly Gaussian transformation. The first interval of pyramid is initial input image. The scale space of an image is defined as a function L(x,y, σ), which is produced from the convolution of a variable-scale Gaussian G(x,y,σ).

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y), \tag{1}$$

$$G(x, y, \sigma) = \frac{1}{2\Pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}, \tag{2}$$

I(x,y) is the value in an input image. After current octave produced, the next octave is down-sampled by a factor of 2. Parameter σ decides the ambiguity of scale space. The ambiguities produced by Gaussian have transmissibility, which means scale space L_i can be produced from L_{i-1} instead of input image.

$$L_i(x, y, \sigma') = G_i(x, y, \sigma') * L_{i-1}(x, y), \tag{3}$$

Feature Points Detection. Feature points are then detected as maxima/minima of the Difference-of-Gaussians (DoG) that in octave*interval pyramid. DoG function D(x,y,σ) is produced from the difference of two nearby scales. After a point is selected, its localization information is computed to check whether it can accurately reflect its nearby data for accurate location, scale, and ratio of principal curvatures. This information allows points to be discarded that have low contrast (and are therefore sensitive to noise) or are poorly localized along an edge.

Description Stage. Description stage contains two steps: orientation assignment and descriptor representation. Each feature point is first assigned one orientation based on local image gradient directions in order to achieve invariance to rotation. Then, it computes a descriptor vector for each feature point such that the descriptor is highly distinctive and invariant to the variations such as illumination, 3D viewpoint, etc. For each feature points, a 128-dimension vector is then created.

2.2 Time Analysis of SIFT

For later analysis, we use Mikolajczyk's image database [7] as input and gather the percentage of execution time for each stage in original sequential SIFT by VTune [11]. The time fraction of each stage is shown in Fig. 1.

As described in section 2.1, there are many scales (intervals) in Gauss pyramid. In accordance with Eq. 3, the computational loads of each scale are different. Moreover, on feature detection stage, as the nether intervals are more notable than upper intervals, more feature points would be detected on nether intervals. Since the workload on each scale is different, it would bring in a challenge of parallelism: difficult to balance tasks. More detail time fraction should be given to help the parallelism implementation. Therefore, we analyze time fraction of each scale of these three stages. The time distribution of gauss pyramid is shown on the left part of Fig. 2 and feature detection is on the right. In Fig 2, the outside triangle represents pyramid structure; each trapezium represents an octave, and inside parallelogram represents interval; the values in them are their execution time proportion respectively.

Since there are hundreds or thousands of detected feature points, it is instinctive to describe these points in parallel. However, such a manner will lead to imbalance computation. In order to illustrate this problem clearly, we also

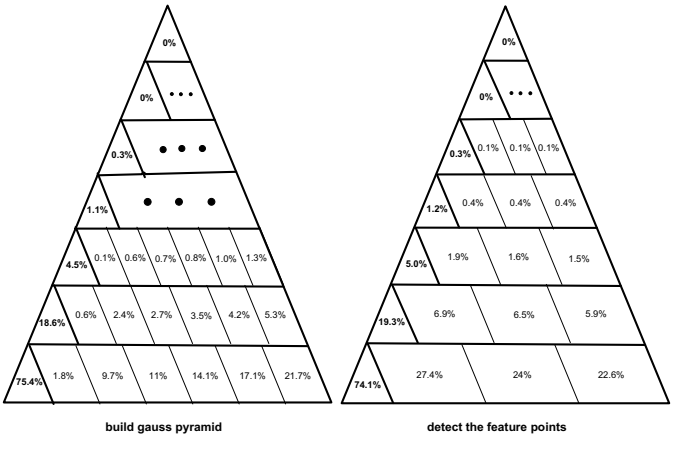


Fig. 2. Time Distribution of Gauss Pyramid (left column) and Time Distribution of Feature Points detection (right column)

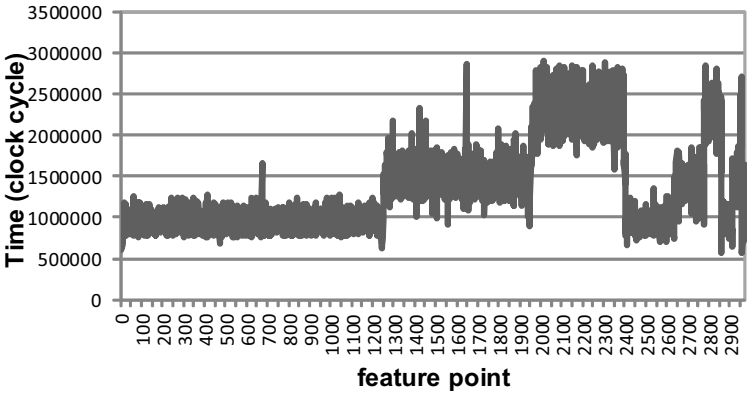


Fig. 3. The computation requirement of each point in the description stage

collect and analyze the time distribution of the feature point description. As the data shown in Fig. 3, the computation of each points are not the same. The reason is that they are computed based on different parameters of DoG. The point with higher value of σ would sample more surrounding points, which has more execution time.

2.3 Parallel Characteristics Analysis

Currently, a variety of parallel hardware architectures, such as multi-core CPU, GPGPU and FPGA, have become more and more popular. Using parallel hardware architecture to accelerate SIFT algorithm can effectively improve its performance. However, before parallelizing SIFT, it is necessary to analyze its parallel characteristics.

- **Parallelism in scale level:** While building Gauss Pyramid, many intervals need to be computed. In original serial SIFT, each interval (Scale) depends on its previous interval, which means it cannot be paralleled.
- **Parallelism in Feature detection:** In the detection stage, each sampled point in pyramid is compared to its 26 neighbors to detect the local maxima/minma of D(x,y,σ). As comparison operations are read-only, each sampled point can be processed in parallel.
- **Parallelism in Description:** In the description stage, as each feature point processed independently, they can be processed in parallel.

3 Parallel SIFT Implementation

We implement various parallelism SIFT according to the parallel characteristics analysis and then analyze their scalability. Currently, we mainly design two parallelism: scale level parallelism and block level parallelism. In our implementation, we also optimize workload assignment by a balance allocation algorithm.

3.1 Scale Level Parallelism

In this part, a scale-level parallel SIFT is designed. To parallelize the computation at scale level, the mainly challenges are two folds: 1) the dependency in gauss pyramid becomes the bottleneck of parallelization; 2) different computational complexity in various scales make work imbalance. To overcome these two problems, we first remove the dependency in gauss pyramid through a new computing approach and then balance the workload on pyramid. We implement two versions to allocate feature points: high coupling method and balance allocation algorithm. The details are given as follows.

– **Remove dependency and balance workload on Gauss Pyramid:** While building gauss pyramid, each interval (scale) depends on the result of its previous interval, which leads to sequential operations for building gauss pyramid. In order to eliminate the dependence, we modify the computing approach of L_i based on the characteristic of gauss computation that continuous k times gauss transform from σ_{i+1}, σ_{i+2}..., to σ_{i+k} equal to direct gauss transform from the kth interval by parameter σ.

$$\sigma^2 = \sigma_{i+1}^2 + \sigma_{i+2}^2 + \sigma_{i+3}^2 + ... + \sigma_{i+k}^2 \tag{4}$$

Based on this characteristic, we modify the process of building gauss pyramid. Since the count of gauss computation of each interval is proportional to that of image points in it, each interval is first assigned a weight value related to its scale. Then the intervals are partitioned into N sets (Note N can be the core number of CMP). The goal of partition is to achieve all the weights of these set are same, which mean a balanced partition. After finishing the partition, one interval in each set is chosen to compute its gauss based on those of the original image (interval 0). And then, the gauss values of other interval in the same set are computed based on its results. In current design of SIFT, there are 16 intervals in the pyramid and the scale of these interval are not the same. Therefore, it is difficult to achieve a balance partition for scale level parallelism while the core number becomes larger, such as 16.

– **Parallelize Description:** As each feature point processed at the same way and independence with each other, they can be processed in parallel. We implement two versions to allocate points to description thread.
  - Method I is high coupling method. To avoid the overhead of creating thread, assigning tasks and thread synchronization, a feature point is detected and described in the same thread. However, as the workloads of computing description are different according to its scale, and the number of feature points on different scales are unequal, this parallel method cannot achieve a balance partition.
  - Method II (which is referred to as *rr* in the experimental part) is a more balance allocation algorithm. According to the analysis in Section 2.2, there are three types of points in computing description based on its located interval. Hence we create three queues to respectively store the detected feature points. As the workloads of feature points in the same

queue are almost equal, the points in the same queue are averagely allocated to different cores in a CMP.

Scalability Analysis. After resolving these challenges, we implement two scale level parallelism versions on multi-core CPU. The scalability of scale level parallelism is mainly constrained two factors. The first one is the number of scale in the SIFT algorithm. Generally, there are only 21 scales in Gauss pyramid. Second, the first interval accounts for about 25% of execution time of building gauss pyramid, which also leads to unbalanced partition. Therefore, it is different to achieve good scalability with the core number increasing.

3.2 Block Level Parallelism

Besides the scale-level parallelism, we also exploit another type of task parallelism, block level parallelism, to speedup SIFT. SIFT treats every pixel in an image as the same, and every pixel is computed independent in SIFT. Therefore, all these pixels can be processed at the same time, which brings in high degree of parallelism. We introduce our parallel scheme based on this parallelism in this section.

As there are too many pixels in an image, computing all these pixels at the same time is impossible. Therefore, we use the block to organize these pixels, and process these blocks at the same time. The main challenge of block level parallelism is how to choose the blocking granularity in each stage: coarser-granularity block or finer-granularity block. The details are given as follow.

- **Gauss pyramid building:** While building gauss pyramid stage, every pixel is processed by the same gauss smooth operation. Moreover, there are a lot of memory operations when computing the gauss pyramid. In order to improve the efficiency of memory operations, we partition the blocks to make its memory continuous.
- **Feature point detection:** If one pixel is at the boundary of a block on the pyramid, it needs to process some additional computation, such as scale localization and boundary checking, which bring in the distinct computational complexity of each block. Moreover, only the detected feature points need to be processed in later stages. So in order to take full advantage of the multi-core computing resource and achieve balanced partition, we choose the finer block partition method to parallel the feature point detection stage, which is similar to those in the scale level parallelism.
- **Feature point description parallelism:** Similar to the scale-level parallelism. We also implement two versions to computing description.
  - Method I is high coupling allocation. Detecting features and describing the features in a block is processed by the same thread.
  - Method II is balance allocation, which is the same as scale-level parallelism.

Overall, the input image first is divided with coarse block partition to build the gauss pyramid. After this, the feature points are detected by fine block partition. At last, we use the two methods like in the scale-level parallelism to describe the feature points.

Scalability Analysis. Unlike scale-level parallelism, block-level parallelism has much better scalability when the threads increased. For the block-level parallelism, there are more blocks than the threads in most case, so when the treads increased, block-level parallelism only need to reallocate the blocks to more threads.

4 Experimental Results

In this section, we evaluation the performance of our parallel SIFT implementations on a real multi-core machine. The machine is a SMP system with 16 cores. Each core is running on 1.6GHz and has a 64KB L1 instruction cache and 64KB L1 data cache. Every two cores share a 2MB unified L2 cache, totally, there are 16MB unified L2 cache in this system.

The parallel SIFT is implemented by pthread library [15], and is compiled by the GCC (GNU C/C++ Compiler) 4.3.2 without optimization argument (-O0) under Debian 5.0 Linux system. We use two image data sets in our experiments: one includes 48 images [7], which is come from 8 images that widely used in computer vision and others are come from these 8 images with kinds of transforms (rotation, illumination and so on). We use this image set to evaluate the parallelism performance of our implementations; another is collected 500 images from Internet. We use this larger image set to finally test parallel effect of our implementation in real system.

4.1 Parallel Performance on 4 Threads

We first evaluate our parallel SIFT implementations on 4 threads. We test all the 4 parallel schemes on SIFT referred in above section use the 48 images data set. The performance improvements are shown in Fig. 4.

In Fig. 4, the scheme marked with **rr** means that the feature point is paralleled by balance allocation algorithm. Based on the results shown in Fig. 4, we get the following observations:

- All the parallel schemes can get a good performance improvement (above 3.5X) on 4 threads parallelism except the original scale parallelism (3.24X). This shows that the schemes introduced in this paper are suitable for the SIFT parallelism.
- The scheme with balance allocation algorithm brings more performance improvement than the original scheme. The scale parallelism with balance allocation algorithm gets a speedup of 3.78X, which is much better than the original scale parallelism with 3.24X. Such a result illustrates that the parallel effect on feature points description is the most important on SIFT parallelism, which can affect the total parallel performance greatly.
- The block parallel scheme with balance allocation algorithm achieves the best 3.94X speedup among all the schemes. This is because the block parallel scheme brings the most balance work allocation at feature point detection compared to scale parallel scheme, and the balance allocation algorithm can get the best allocation at feature point description.

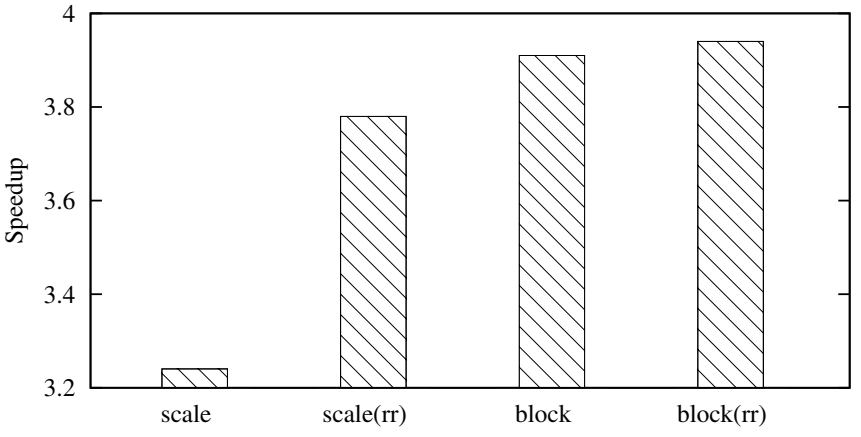


Fig. 4. Parallel performance of 4 threads

4.2 Block Size Analysis

There is a very important argument that will also affect the block parallel performance, which is the block size. If the block size is too large, the block work may be allocated unbalance, this will make some threads get much more work than others, and these threads will become the bottleneck of the program. If the block size is too small, the cache locality will be broken by the threads, this will bring extra cache miss, also reduce the parallel performance. Therefore, choosing a suitable block size can make the block parallel scheme achieves the best parallel results. In order to find the best block size, we test the 48 images data set based on original block parallel scheme with different block size on 4 threads. The results are shown in Fig. 5.

As the data shown in Fig. 5, 16 * 16 block size can get the best scalability. The main reason of this result is that the 16 core machine uses the cache line with 64 bytes size. The 16 float point is taken just 64 bytes memory, which is equal to a cache line size, so 16 * 16 block size can get a best balance between cache locality and work allocation. Therefore, in our following test, an image is partitioned into 16*16 blocks while it needs to be partitioned into blocks.

4.3 Scalability Analysis

Our parallel SIFT implementation achieves an ideal speedup on 4 threads. In order to know whether the parallel schemes are suit for the more threads parallelism, we test the scalability of our parallel SIFT implementation on 8 threads and 16 threads. The results are shown in Fig. 6.

Based on the results shown in Fig. 6, we can find that the performance of scale parallel scheme is the worst, and the two versions of block parallel scheme are all better than the two versions of scale parallel scheme. The major reason

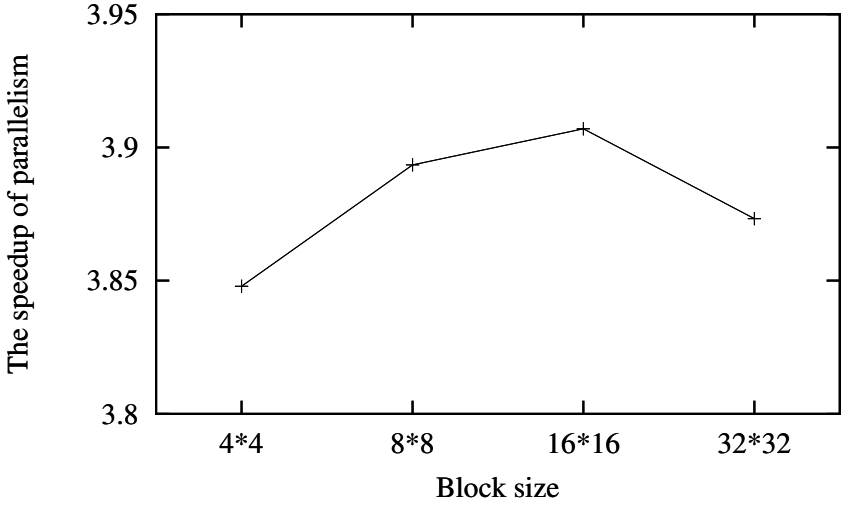


Fig. 5. Parallel performance with different block size on 4 threads

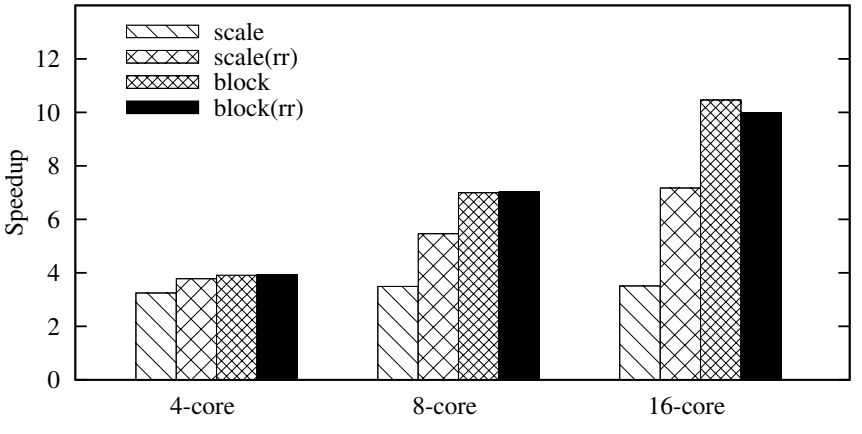


Fig. 6. Parallel performance scalability

of such a result is that the large scales in the first octave take up over 20% work, which leads to unbalanced partition for scale level parallelism.

Although block parallel scheme with balance allocation algorithm has a better performance on 4 threads, it is slower than the original block parallel scheme on 16 threads.This is because there have to be synchronization between detection and description stages when balance allocation is added, which cancel out some part of the performance gained from balance workload. When the number of threads is scaled up to 16, the overhead of synchronization is larger than the performance gained from balance allocation.

The results in Fig. 6 also indicate that the speedups of all the parallel schemes are farther from the ideal speedup when the threads increase. The best scheme of original block parallel achieves only 10.46X speedup on 16 threads. This is mainly because the percentage of un-parallelized and poor-parallelized part in SIFT will increase as the threads increase. These parts in SIFT will lead to a large reduction of the speedup on parallel SIFT implementation.

4.4 Real Image Evaluation

After the previous experiments, we find that the block parallel scheme with balance allocation algorithm achieves the best parallel performance on 4 threads, and on 16 threads the original block parallel scheme achieves the best. The best schemes can get 3.94x speedup on 4 threads and 10.46x speedup on 16 threads by using the 48 images data set. Even the 48 images are widely used, but the real images may be much more complex. In order to check the parallel performance on the real images, we test the block parallel scheme by using 500 images from Internet. Fig. 7 depicts the test result.

From Fig. 7, we see that the block parallel scheme gets 10.08x speedup, which is a little less than the performance of 48 images. We check the characteristic of two images data sets and find that the 48 images data set average contains 2701 feature points, which is much more than 1500 feature points of 500 real images. This makes the feature points description stage that has most parallel improvement takes less time percentage, and then leads to less speedup than that of 48 images data set.

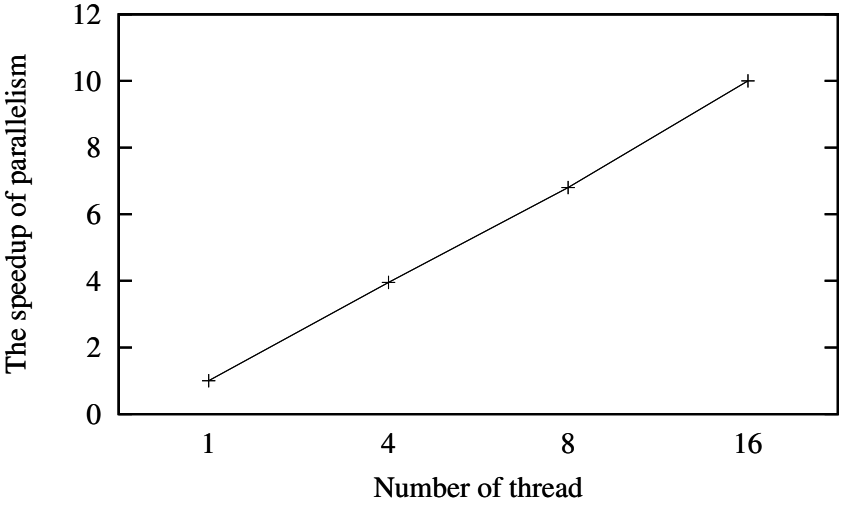


Fig. 7. The best parallel performance on real images

5 Related Work

In this section, we will introduce the local feature-based algorithms and discuss the acceleration works on SIFT.

5.1 Image Local Feature-Based Extraction Algorithms

Besides SIFT, there are several algorithms to extract the local features from images. SURF [20] is another widely used local feature-based algorithm. It can run faster than SIFT and achieves almost the same accuracy. RIFT [21] is the rotation invariance version of SIFT, which can get better rotation and affine invariance. GLOH [22] enhances the specificity and robustness of the descriptor on SIFT. PCA-SIFT [23] uses the Principal Component Analysis (PCA) to reduce the complexity of descriptor. Hence, it can achieve faster speed than SIFT with little accuracy reduce.

5.2 Accelerating SIFT Algorithm

In order to make SIFT more applicable, there are some researches focus on accelerating SIFT algorithm via multi-core processors. Zhang et. al. [25] explored the parallelization of SIFT algorithm and proposed a improved parallelization method to get better speedup. They show a 6.2x parallelization speedup on an 8-core machine. Feng et. al. [24] implemented a parallelization algorithm to a 16-core machine and analyzed the factors that affect the SIFT parallelization, such as cache size, thread communications and so on. Their experiments show a 10-11X speedup. There are also several proposed acceleration techniques on GPGPU to make SIFT run in real-time. Sinha et. al. [26] implemented SIFT algorithm on GPGPU, and the results show that the SIFT algorithm can extract feature points from 640 * 480 images at 10FPS on a GeForce 7800 GTX. Heymana et. al. [27] presented a GPGPU-accelerated SIFT implementation that achieved 20FPS on QuadroFX 3400.

Seth et. al. [28] explored the parallelization implementation of SIFT algorithm on large images. They found that SIFT requires a lot of memory when processing large images, so their implementation on 8-core processor only gets a 2x speedup which is limited by the cache behavior. On GPGPU FX5800, their implementation achieves a 13x speedup over their CPU implementation.

Different from the above SIFT algorithm accelerations, we first analyze different parallelism in SIFT, which is generally ignored by those previous algorithms. And then, we select the best one to parallelizing SIFT, which leads to better performance improvement than those previous ones.

6 Conclusion and Future Work

In this paper, we first analyzed the parallel characteristics of state-of-the-art local based image feature algorithm SIFT. Then we implemented two types of parallelism SIFT according to the parallel characteristics analysis, including scale level

parallelism and block level parallelism. In scale level parallelism, we eliminate the dependency on build gauss pyramid stage and balance the tasks according to workload of interval. On description stage, we implement two points assignment method: high coupling method and balance allocation algorithm. In block level parallelism, we parallelize the SIFT by the idea of blocking, and we also implement two points assignment method. As the experimental results shows, all the parallel implementation can get a good performance improvement (above 3.5X) on 4 threads parallelism except the original scale parallelism (3.24X). The block parallel scheme with balance allocation algorithm achieve a speedup by a factor of 3.94x. This is because compared with scale parallel scheme, the block parallel scheme brings the most balance work at feature points detection, and it get the best allocation at feature points description by the balance allocation algorithm. We also evaluate the scalability of our parallel SIFT implementation on 8 threads and 16 threads. As balance allocation algorithm brings not only the balance workload but also the synchronization overhead between detection and description stages, when the number of threads is scaled up to 16, block parallel scheme without balance allocation get better performance. The block parallel scheme can achieves 10.46x speedup on 16 threads, which has a good scalability to meet the real-time applications demand.

Acknowledgments. This work was funded by China National Natural Science Foundation under grant numbered 60903015, a joint program between China Ministry of Education and Intel numbered MOE-INTEL-10-04, Fundamental Research Funds for the Central Universities in China and Shanghai Leading Academic Discipline Project (Project Number: B114).

References

1. Bohn, R., Short, J.: How much information? 2009, report on american consumers, San Diego, CA (December 2009)
2. Marques, G.B.B.O., Mayron, L.M., Gamba, H.R.: An attention-driven model for grouping similar images with image retrieval applications. EURASIP J. of Applied Signal Processing 2007(1), 116 (2007)
3. Smeulders, A.W.M., Member, S., Worring, M., Santini, S., Gupta, A., Jain, R.: Content-based image retrieval at the end of the early years. IEEE Transactions on Pattern Analysis and Machine Intelligence 22, 1349–1380 (2000)
4. Lowe, D.G.: Object recognition from local scaleinvariant features. In: Computer Vision, vol. 2, pp. 1150–1157 (1999)
5. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision 60(2), 404–417 (2004)
6. Mikolajczyk, K., Schmid, C.: A performance evaluation of local descriptors. IEEE Transactions on PAMI 27(10), 1615–1630 (2005)
7. Mikolajczyk, K.: Local feature evaluation dataset, http://www.robots.ox.ac.uk/~vgg/research/affine/
8. Bay, H., Tuytelaars, T., Van Gool, L.: SURF: Speeded up robust features. In: Leonardis, A., Bischof, H., Pinz, A. (eds.) ECCV 2006. LNCS, vol. 3951, pp. 404–417. Springer, Heidelberg (2006)

9. Tang, F., Gao, Y.: Fast near duplicate detection for personal image collections. ACM Multimedia, 701–704 (2009)
10. Wu, X., Ngo, C.-W., Li, J., Zhang, Y.: Localizing volumetric motion for action recognition in realistic videos. ACM Multimedia, 505–508 (2009)
11. Intel. Intel vtune performance analyzer, http://software.intel.com/en-us/intel-vtune/
12. ISO/IEC/JTC1/SC29/WG11, C.D.: 15938-3 MPEG-7 Multimedia Content Description Interface - Part 3. MPEG Document W3703 (2000)
13. Huang, J., Kumar, S.R., Mitra, M., Zhu, W.J., Zabih, R.: Image Indexing using Color Correlograms. In: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition, pp. 762–768. IEEE Computer Society, Los Alamitos (1997)
14. Wu, P., Ro, Y.M., Won, C.S., Choi, Y.: Texture Descriptors in MPEG-7. In: Skarbek, W. (ed.) CAIP 2001. LNCS, vol. 2124, pp. 21–28. Springer, Heidelberg (2001)
15. Nichols, B., Buttlar, D., Farrell, J.P.: Pthreads Programming. O'Reilly & Associates, Inc., Sebastopol (1996)
16. Wan,Y., Yuan, Q., Ji, S., He, L., Wang, Y.: Intel. Intel icc compiler, http://software.intel.com/en-us/intel-compilers/
17. Wang.: A survey of the image copy detection. In: IEEE Conference on Cybernetics and Intelligent Systems (2008)
18. Berrani, S., Amsaleg, L., Gros, P.: Robust content-based image searches for copyright protection. In: Proceedings of ACM Workshop on Multimedia Databases (2003)
19. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features, pp. 511–518 (2001)
20. Bay, H., Tuytelaars, T., Van Gool, L.: SURF: Speeded up robust features. In: Leonardis, A., Bischof, H., Pinz, A. (eds.) ECCV 2006. LNCS, vol. 3951, pp. 404–417. Springer, Heidelberg (2006)
21. Lazebnik, S., Schmid, C., Ponce, J.: A sparse texture representation using local affine regions. Technical Report CVR-TR-2004-01, Beckman Institute, University of Illinois (2004)
22. Mikolajczyk, K., Schmid, C.: A performance evaluation of local descriptors. IEEE Transactions on Pattern Analysis and Machine Intelligence 10(27), 1615–1630 (2005)
23. Ke, Y., Sukthankar, R.: PCA-SIFT: a more distinctive representation for local image descriptors. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, vol. 2, pp. 506–513 (2004)
24. Feng, H., Li, E., Chen, Y., Zhang, Y.: Parallelization and characterization of sift on multi-core systems. In: IISWC 2008, pp. 14–23 (2008)
25. Zhang, Q., Chen, Y., Zhang, Y., Xu, Y.: Sift implementation and optimization for multi-core systems. In: Parallel and Distributed Processingv (IPDPS), pp. 1–8 (2008)
26. Sinha, S., Frahm, J.-M., Pollefeys, M., Genc, Y.: Feature tracking and matching in video using programmable graphics hardware. In: Machine Vision and Applications (2007)
27. Heymann, S., Muller, K., Smolic, A., Froehlich, B., Wiegand, T.: SIFT implementation and optimization for general-purpose GPU. In: WSCG (2007)
28. Warn, S., Emeneker, W., Cothren, J., Apon, A.: Accelerating SIFT on Parallel Architectures. In: Cluster Computing and Workshops(CLUSTER), pp. 1–4 (2009)

Modality Conflict Discovery for SOA Security Policies

Bartosz Brodecki, Jerzy Brzeziński, Piotr Sasak, and Michał Szychowiak

Poznań University of Technology
Piotrowo 2, 60-965 Poznań, Poland
{BBrodecki,JBrzezinski,PSasak,MSzychowiak}@cs.put.poznan.pl

Abstract. This paper considers the problem of modality conflicts in security policies for Service-Oriented Architecture (SOA) environments. We describe the importance of this problem and present an algorithm for discovering modality conflicts with low overhead. Often being of large scale and compound structure, SOA systems can definitely benefit from that efficiency boost. Another advantage of the proposal over previously developed algorithms is its formal proof of correctness, also presented in this paper.

1 Introduction

Security policy has become indispensable for the management of security restrictions and obligations in modern distributed computing systems, and especially in service oriented environments (i.e. complying to Service-Oriented Architecture model, SOA). Typically, the policies can grow very large (in the number of policy rules) and suffer from conflicts, heavily influencing the correctness policy execution. As an example of such conflicts, in this paper we consider modality conflicts [1]. Informally, modality conflict arise when for any single access operation two or more policy rules lead to ambiguous access control decisions.

Several modality conflict detection algorithms considered in literature use a naive approach of comparing all pairs of policy rules [1,2]. This approach, being inefficient, can be improved by reducing the number of necessary comparisons. Ehab S. Al-Shaer and Hazem H. Hamed present [3] a modality conflict discovery algorithm for firewall policies which uses a specific tree structure to represent all rules, significantly reducing the search space. Their work, however, applies to policy rules for firewalls only.

F. Baboescu and G. Varghese developed another interesting algorithm for detecting modality conflicts [4]. They also analyse only firewall rules and use a binary tree to model hierarchy of IP addresses (separately for source and for destination addresses). Binary vectors are created in nodes of the binary tree to represent relations between source (or destination) addresses of firewall policy rules which use them. Conflict discovery consists in finding common parts of two vectors (one for source address and one for destination address). This algorithm, although very efficient, can only be used in numerical (with natural hierarchy) parameters of policy rules (like IP addresses).

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 112–126, 2011.

Jorge Lobo, Emil Lupu et al. [5] investigate a first–order predicate language to model and analyse general security policies. They use abductive reasoning to estimate if a security policy is free of modality conflicts. The actual conflict detection requires conducting the reasoning separately for each rule, one at a time. Due to the use of a specific policy language, the time complexity of this detection is polynomial with respect to the size of the policy. However, this approach does not really support the conflict resolution, since the result of reasoning states only whether the considered rule causes any conflict or not. It does not reveal which other policy rules are in conflict with the considered one.

Since the existing proposals are inefficient or have limited applicability, searching for a universal and low-cost solution for SOA security policies is strongly motivated.

In this work, we propose such a solution for compound policy rules which offers a polynomial time complexity. Moreover, the proposal is formally proven, and only that can make it truly useful for the policy management.

Recently, the ORCA language [6] has been proposed to fully express a security policy of the SOA environment. It has several SOA-oriented advantages over previously existing policy languages, thus we have decided to base this work on that language and its policy execution mechanisms.

This paper is organized as follows. The next Section briefly overviews the modality conflict problem in SOA security policies. In Section 3 we introduce a more sophisticated algorithm which is based on a generalisation of the algorithm [4] for firewall policies. We prove the correctness of our proposal in Subsection 3.4. Finally, Section 4 gives some insights on further research, concluding this paper.

2 System Model

Let policy $\hat{\mathbb{P}}$ be represented as a set $\{R1, R2, ..., Rk\}$ of policy rules. Every policy rule $Ri(\mathrm{S}i,\mathrm{T}i,\mathrm{A}i,\mathrm{C}i)=\mathrm{D}i$ is composed of a *subject* Si, a *target* Ti, *actions* Ai, *conditions* Ci, and a *decision* Di. Typically, the policy will be defined in an environment in which some hierarchical relations exist between the policy principals (i.e. subjects and targets). Usually, the principals will belong to user groups, roles or any other aggregations. For example, Role1 may *include* User1, among other users, thus Role1 is higher in the principals hierarchy than User1.

In general, modality conflicts are known to arise when two or more policy rules refer to the same subjects and targets [1] but lead to conflicting policy decisions about the allowed actions. In SOA, this problem appears when making a decision about allowing or denying accesses to services (as targets). The policy subject can represent a single user or a client application, a user role (very similar to a group of users) or an aggregated set of those subjects. The target can be a single service, an address group of multiple services or an aggregated set of targets. Also, policy actions are represented as sets of elementary operations (e.g. read, write, invoke). Finally, each condition is represented by a set of pairs: a condition type and its value (e.g., time={weekend 8:00–15:00}). Therefore,

the main difficulty of conflict detection in SOA lies in dealing not with single elements but with the sets of elements, in a typical case.

Definition 1. *Modality conflicts are inconsistencies in the SOA policy specification, which arise when for two rules with opposite decisions there exists intersection of subjects, targets, actions, and conditions.*

Let us consider a sample policy $\hat{\mathbb{P}}$ which actually includes some modality conflicts. Listing 1.1 presents the policy rules in the ORCA language, while Listing 1.2 shows the formal representation of the policy.

Listing 1.1. Sample security policy $\hat{\mathbb{P}}$

```
Role1 can access www.domain.net/service1/* for {read, write},
 if time=8:00-16:00.
Role2 cannot access www.domain.net/* for {write},
 if time=12:00-18:00.
```

Listing 1.2. Security policy $\hat{\mathbb{P}}$ in a formal representation

```
1  R1(S1, T1, A1, C1)=Accept.
2  R2(S2, T2, A2, C2)=Deny.
3  S1={Role1}. // subjects S1 and S2
4  S2={Role2}.
5  T1={"www.domain.net/service1/*"}.    // targets
6  T2={"www.domain.net/*"}.
7  A1={read, write}.    // allowed actions
8  A2={write}.
9  C1={time=8:00-16:00}.    // conditions
10 C2={time=12:00-18:00}.
```

As you can see from Listing 1.2, there are two rules R1 and R2 (lines 1 and 2) giving different decisions. Both rules use different subjects (S1, S2), targets (T1, T2), actions (A1, A2) and conditions (C1, C2). Later, we have a declaration of S1 as composed of Role1 and S2 composed of Role2. Fig. 1 presents possible hierarchy of subjects (H_S) and targets (H_T) for this policy.

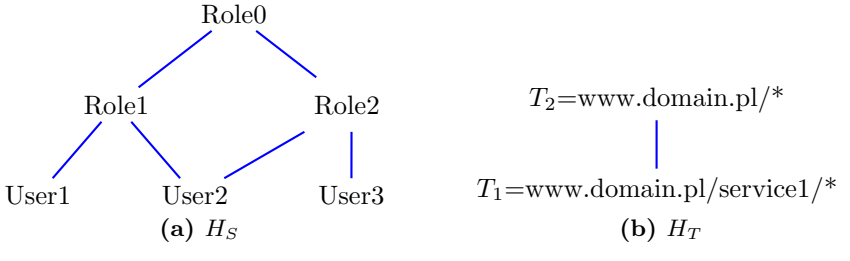


Fig. 1. Principals hierarchy of the sample policy (Listing 1.2)

3 Modality Conflict Discovery for SOA Policies

In this Section, we will describe a modality conflict discovery algorithm supporting any policy rules with condition predicates.

The Improved Modality Conflict Detection algorithm (IMCD) is aimed at reducing the number of necessary comparisons of policy rule pairs. IMCD will use a graph representation of policy principals to initially select rules that might be in a *potential* conflict with any others. Only those rules will be further compared against an actual modality conflict. The expected gain over the naive approach is increased efficiency, as the cost of building the necessary graph will be in general smaller then the cost of full comparison of policy rules (i.e. $O(n^2)$).

3.1 Prerequisites

The conflict detection algorithm must be provided with knowledge of the hierarchy of subjects (H_S) and targets (H_T).

Definition 2. *Principal descendance. When there exist two principals (P1, P2) and one of them (P1) includes the other (P2), then the second principal is directly descendant from the first one (denoted* $P1 \rightarrow P2$*).*

Definition 3. *Priviledges inheritance. Privileges of principal P1 are inherited by principal P2, if there exists principal descendance* $P1 \rightarrow P2$*.*

Policy Object Graph (POG). IMDC algorithm must be provided with the knowledge about the hierarchy of all principals in order to detect all potentially conflicting rules. Policy Object Graph is a directed graph (tree) $\overrightarrow{POG}$ representing that hierarchy. The root of the $\overrightarrow{POG}$ is wildcard any ("*"). Arcs represent the direct principals descendance, i.e. an arc from Px to Py exists in $\overrightarrow{POG}$, if $Px \rightarrow Py$ according to Definition 2. Additionally, by $\overline{POG}$ we will denote a undirected instance of $\overrightarrow{POG}$. The $\overline{POG}$ will be further used only for cycle detection. We will refer to both representations altogether, simply as POG.

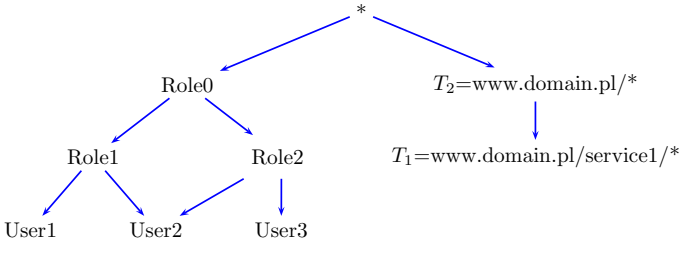


Fig. 2. POG for the hierarchy from Fig. 1

Definition 4. *Transition of principal descendance. If path* $\overline{P}$ *from a node* (Px) *into another node* (Py)*, exists in directed* $\overrightarrow{POG}$*, such as* $Px \rightarrow P_i \rightarrow P_{i+1} \rightarrow ... \rightarrow P_{i+k} \rightarrow Py$*, we say that* Py *is* ***indirectly descendant*** *of* Px*.*

Definition 5. *A principal cycle in the POG exists only if there are at least two paths having only two common nodes (starting and ending node), i.e.:*

$$
\begin{aligned}
&If\, f\, \exists paths\, \overline{P}_i, \overline{P}_j,\ nodes\, Px, Py: \\
&\qquad \overline{P}_i : Px \rightarrow P_i \rightarrow P_{i+1} \rightarrow ... \rightarrow P_{i+k} \rightarrow Py \\
&\quad and\, \overline{P}_j : Px \rightarrow P_j \rightarrow P_{j+1} \rightarrow ... \rightarrow P_{j+m} \rightarrow Py \\
&\qquad\qquad and\, \{P_i, P_{i+1}, ..., P_{i+k}\} \cap \{P_j, P_{j+1}, ..., P_{j+m}\} = \oslash
\end{aligned}
$$

then, there exists a cycle in the $\overline{POG}$*, which includes all the nodes from both paths* $\overline{P}_i$ *and* $\overline{P}_j$*.*

Remark, for the hierarchy in Fig. 2 one principal cycle exists, including the following nodes: Role0, Role1, User1, Role2.

For some reason, that will become clear later in this paper, we distinguish a special case of the POG cycle.

Definition 6. *Inheritance cycle is a cycle in the POG, which does not include any nodes representing users.*

Remark, in Fig. 2 there is not any inheritance cycle, because principal User1 is a user.

3.2 Preparation Phase

The IMCD algorithm requires a preparation phase (Alg. 1) which creates the necessary data structures. The first step of the preparation phase involves creating of *POG*. Actually, two POG graphs are created: one for all the subjects — POG_S, and the other for all the targets — POG_T. Then, all the nodes of both POGs are extended with binary vectors of the size equal to the number of policy rules. The time complexity of this phase linearly depends on the number of policy rules and the size of the principals hierarchy.

The subject of i-th policy rule (Ri), referred to further as Ri.Subject, is represented in POG_S as a single node. With that node the IMDC algorithm maintains binary vector BV such that $BV[i] \leftarrow 1$ (reflecting the fact that the subject is used by the i-th rule).

Similar vectors for the targets of all rules are maintained in the POG_T graph.

Propagation of BV in POG_S*.* All BV will propagate their values down the POG_S graph to the descendant nodes. Moreover, for each inheritance cycle in POG_S, BV from all the nodes belonging to that cycle will binary summate to reflect the existence of the inheritance cycle.

Propagation of BV in POG_T. In POG_T, the BV will propagate in two directions (up and down the graph). Therefore, each POG_T node will maintain three different types of BV:

1. For the rule target of a given POG node (denoted as BV).
2. For descending propagation of the BV down the $\overrightarrow{POG}$ graph ($\overrightarrow{BV}$).
3. For ascending propagation of the BV up the $\overrightarrow{POG}$ ($\overleftarrow{BV}$).

Propagation of vectors through an inheritance cycle in the POG_T graph will summate $\overrightarrow{BV}$ vectors in the same way as propagation of BV vectors in the POG_S graph.

3.3 Modality Conflict Detection Algorithm

For all policy rules IMCD looks for three BV, one for subject (BV_S) and two for target ($\overrightarrow{BV}_T$ and $\overleftarrow{BV}_T$). BV_S is the BV stored in the POG_S node, which represents the subject of rule Ri, i.e.$BV_S \leftarrow POG_S(Ri.Subject).BV$. Similarly, $\overleftarrow{BV}_T \leftarrow POG_T(Ri.Target).\overleftarrow{BV}$ and $\overrightarrow{BV}_T \leftarrow POG_T(Ri.Target).\overrightarrow{BV}$. These three BV will be used to check which rules potentially conflict with Ri. IMCD algorithm checks BV to find out which rules reflected in BV_S are also reflected in $\overrightarrow{BV}_T$ or in $\overleftarrow{BV}_T$. These rules create a ***set of potential conflicts*** and will be then processed by naive comparison algorithm (a revised version of NaMCD denoted hereafter NaIMCD).

IMCD makes only a limited use of the naive approach in procedure Compare (Alg. 2). This procedure will only compare actions and conditions of those rules which have been formerly suspected as a potential conflict. Shall any of these comparisons give a negative result, the compared rules are definitely not conflicting. Otherwise, these rules are detected as being in an actual modality conflict.

Main part of the detection. The main part of the IMCD algorithm is presented in Alg. 3. It takes two arguments. The first one is policy $\hat{\mathbb{P}}$, i.e. a set of n rules (Ri, where i=1..n). A sample policy was presented in Listing 1.2. The second argument is the hierarchy of all principals (H_S and H_T) needed for creating POG. The time complexity of this phase is linearly depending on the policy size, i.e. $O(n)$.

3.4 Correctness Proof

Among all possible relations between policy principals P_1, P_2 (considered in [7] for instance) we distinguish a dependency relation important from a modality conflict perspective.

Definition 7. *A hierarchical dependency of principals (subjects or targets) may be one of the following:*

```
1: function Preparation(Policy {R1, ..., Rn}, Hierarchy {H_S, H_T})
2:     POG ← Create a graph from H_S, H_T
3:     POG_S ← POG    :: All BV in POG are empty
4:     POG_T ← POG
5:     for i ← 1..n do
6:         POG_S(Ri.Subject).BV[i] ← 1    :: Set BV[i] in node Ri.Subject
7:         POG_T(Ri.Target).BV[i] ← 1
8:     end for
9:     foreach S in POG_S do    :: descending propagation
10:        if POG_S(S).BV ≠ ∅ then
11:            foreach direct descendant D of S do    :: i.e. ∀D: S → D
12:                POG_S(D).BV ← POG_S(D).BV ∪ POG_S(S).BV
13:            end for
14:        end if
15:    end for
16:    foreach T in POG_T do    :: descending propagation
17:        if POG_T(T).BV ≠ ∅ then
18:            foreach direct descendant D of T do
19:                POG_T(D).BV→ ← POG_T(D).BV ∪ POG_T(T).BV ∪ POG_T(D).BV→
20:            end for
21:        end if
22:    end for
23:    foreach T in POG_T do    :: ascending propagation (from leaves to the root)
24:        if POG_T(T).BV ≠ ∅ then
25:            foreach parent D of T do
26:                POG_T(D).BV← ← POG_T(D).BV ∪ POG_T(T).BV ∪ POG_T(D).BV←
27:            end for
28:        end if
29:    end for
30:    foreach cycles c ∈ ℂ do    :: all BV in a cycle must be the same
31:        commonBV_S ← ∅
32:        commonBV_T ← ∅
33:        foreach node n ∈ c do
34:            commonBV_S ← commonBV_S ∪ POG_S(n).BV
35:            commonBV_T ← commonBV_T ∪ POG_T(n).BV→
36:        end for
37:        foreach node n ∈ c do
38:            POG_S(n).BV ← commonBV_S
39:            POG_T(n).BV→ ← commonBV_T
40:        end for
41:    end for
42:    return POG_S and POG_T
43: end function
```

Alg. 1. Preparation phase of the IMCD algorithm

1. *Equality (type DT1): Both principals are literally the same.*
2. *Membership (DT2): One principal includes the second one (which can make a direct or indirect principal descendance).*
3. *Inheritance cycle (DT3): Both principals belong to the same inheritance cycle.*

Please observe that two principals $P1$, $P2$ are hierarchically dependent when $P1 \cap P2 \neq \emptyset$.

```
 1: procedure Compare(Rule R, potentialConflictSet {R1, ..., Rk})
 2:     for i ← 1..k do
 3:         if R ≠ Ri then
 4:             if R.decision ≠ Ri.decision then
 5:                 if CompareA(R.action, Ri.action)
 6:  and CompareC(R.condition, Ri.condition) then
 7:                     ConflictSet ← ConflictSet ∪ {(R, Ri)}
 8:                 end if
 9:             end if
10:         end if
11:     end for
12: end procedure
13: function CompareA(A1, A2)    :: A1 and A2 are sets of actions
14:     if A1 ∩ A2 ≠ ∅ then
15:         return true
16:     end if
17:     return false
18: end function
19: function CompareC(C1, C2)    :: C1 and C2 are sets of conditions
20:     c ← condition_predicate
21:     foreach condition_predicate_type c ∈ C1 do
22:         if c ∈ C2  and C1[c] ∩ C2[c] = ∅ then
23:             return false
24:         end if
25:     end for
26:     return true
27: end function
```

Alg. 2. Pseudocode of procedure Compare

From Definition 1 a modality conflict between two rules occurs when all the following conditions hold:

1. Subjects of both rules are hierarchically dependent and targets are hierarchically dependent.
2. Actions of both rules have common parts and conditions also have common parts.
3. Decisions are contrary.

```
1: procedure IMCD(Policy ℙ̂, Hierarchy {H_S, H_T})
2:     ConflictSet ← ⊘
3:     for Policy ℙ̂ = {Ri where i = 1..n} do
4:         Graph{POG_S, POG_T} ← Preparation(ℙ̂, {H_S, H_T})
5:         for j ← 1..n do
6:             ConflictDetect(Graph {POG_S, POG_T},Rule Rj)
7:         end for
8:     end for
9: end procedure
10: procedure ConflictDetect(Graph {POG_S, POG_T}, Rule R)
11:     BV_S ← POG_S(R.Subject).BV
12:     BV→_T ← POG_T(R.Target).BV→
13:     BV←_T ← POG_T(R.Target).BV←
14:     BV_1 ← BV_S ∩ BV→_T
15:     BV_2 ← BV_S ∩ BV←_T
16:     BV ← BV_1 ∪ BV_2
17:     PotentialConflictSet ← ⊘
18:     if Numbers of bits set in BV ≥ 2 then
19:         PotentialConflictSet ← {Ri :: BV[i] == 1, where i = 1, n}
20:         Compare(R, PotentialConflictSet)    :: Use Alg. 2
21:     else
22:         No Conflict
23:     end if
24: end procedure
```

Alg. 3. Main part of the IMCD algorithm

Initially, IMCD will try to detect all the pairs (or more precisely – tuples, since there can be more rules in the mutual conflict) of *potentially conflicting rules* for which the first condition holds (i.e. $S1 \cap S2 \neq \oslash \wedge T1 \cap T2 \neq \oslash$). Then, in the set of such potentially conflicting rules, we will compare each pair and exclude those pairs which do not fulfill the two latter conditions. The remaining rules are finally considered in a modality conflict.

Thus, the IMCD algorithm will use the BV to check for potentially conflicting rules.

Theorem 1. *Algorithm IMCD discovers all pairs of potentially conflicting rules in a security policy.*

In order to prove the Theorem 1 we introduce the following Lemmas.

Let us consider two rules R1(S1,T1,A1,C1)=D1 and R2(S2,T2,A2,C2)=D2.

Lemma 1. *Algorithm IMCD is able to detect any type of hierarchical dependencies (i.e. DT1, DT2, DT3) between the subjects of two rules.*

Proof. Let us assume that there exists a hierarchical dependency between subjects S1, S2 of two rules R1, R2. For the simplicity of presentation, let R1 and R2 be the only rules in the policy (thus, the size of BV vectors will be 2). We

will show, that this dependency will be undoubtedly reflected by BV_S vector in procedure ConflictDetect of the main detection phase.

First, suppose that S1 and S2 are literally the same (S1=S2), we will denote it simply as S. Let $POG_S(S).BV$ be the binary vector of node S in $\overrightarrow{POG_S}$ in the preparation phase of the IMCD algorithm. According to line 6 of Alg. 1 $POG_S(S).BV[1] \leftarrow 1$ and $POG_S(S).BV[2] \leftarrow 1$, as S appears in both $R1$ and $R2$. This appearance will be reflected in the main detection phase (Alg. 3) when procedure ConflictDetect processes rule R1. Precisely, it will be reflected by vector $BV_S \leftarrow POG_S(S).BV = [11]$ (set in line 11) further used in deciding about the content of *PotentialConflictSet* (line 18). Thus, the knowledge about a hierarchical dependency DT1 between subjects of rule $R1$ and $R2$ will not be lost. Actually, when the main detection phase processes rule R2, the dependency between subjects will be the detected once again. Therefore, type DT1 of hierarchical dependencies between subjects of two rules will be found twice.

Next, let us consider dependency DT2 between subjects S1 and S2.

1) Suppose that S2 is directly descendant from S1 ($S1 \rightarrow S2$). Then, in the $\overrightarrow{POG_S}$ graph an arc exists from S1 to S2. $POG_S(S1).BV$ is the binary vector of node S1 created in $\overrightarrow{POG_S}$ in the preparation phase of the IMCD algorithm. Similarly, $POG_S(S2).BV$ is the binary vector of node S2. According to line 6 of Alg. 1 $POG_S(S1).BV[1] \leftarrow 1$ and $POG_S(S2).BV[2] \leftarrow 1$. Since S1 has a direct descendant S2 then the propagation of BV makes $POG_S(S2).BV \leftarrow POG_S(S2).BV \cup POG_S(S1).BV$ (lines 9 – 15 of Alg. 1) which results in $POG_S(S2).BV \leftarrow [01] \cup [10] = [11]$. The main detection phase (Alg. 3) starts with processing rule R1 (line 5), and according to line 11, $BV_S \leftarrow POG_S(S1).BV = [10]$ then subject S1 appears only in rule $R1$ (because in BV_S bit is set only in the first place). Next, the detection phase processes rule R2, and this time $BV_S \leftarrow POG_S(S2).BV = [11]$. As both, the first and second bit of BV_S is set, IMCD realizes the dependency of subjects of the first and the second rule (S1 and S2).

2) Suppose that S2 is indirectly descendant from S1. Then, according to lines 9 to 15 of Alg. 1, every direct descendant Sx of $S1$ will sum its $POG_S(Sx).BV$, with $POG_S(S1).BV$ and propagate the result to its descendants. And every descendant will do the same with its own BV. In consequence, S2 will finally sum $POG_S(S2).BV$ with the BV propagated from its ascendants. In this new $POG_S(S2).BV$, the bits corresponding to both rules $R1$ and $R2$ will be set. Then, when the main detection phase starts with processing rule R2, then a dependency between S1 and S2 will be found.

We have shown in above points that the IMCD algorithm will detect dependency DT2 between subjects S1 and S2.

Finally, assume hierarchical dependency DT3 (an inheritance cycle) between S1 and S2.

Suppose that S1, S2, Sx and Sy form the inheritance cycle in the $\overline{POG_S}$ graph (Fig. 3). According to line 6 of Alg. 1 $POG_S(S1).BV[1] \leftarrow 1$ and $POG_S(S2).BV[2] \leftarrow 1$. At the end of the preparation phase all the nodes in the cycle will have the same BV, the sum of all BV from these nodes (line 34 of

Alg. 1), which results in $commonBV_S = [11]$. The main detection phase (Alg. 3) will set $BV_S \leftarrow POG_S(S1).BV = [11]$ for R1 (line 11). Because the first and the second bit of BV_S are set, IMCD sees the dependency between subject S1 of rule $R1$ and subject S2 of the second rule $R2$. Therefore, also type DT3 of hierarchical dependencies between subjects will be found.

We have shown in the above steps that IMCD algorithm will inevitably detect any dependencies (DT1, DT2, DT3) between subjects S1 and S2 of rules R1 and R2. Q.e.d.

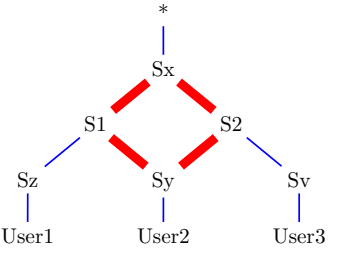


Fig. 3. Sample inheritance cycle in $\overline{POG_S}$

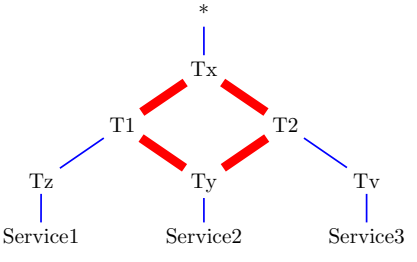


Fig. 4. Sample inheritance cycle in $\overline{POG_T}$

Lemma 2. *Algorithm IMCD is able to detect any type of hierarchical dependencies (i.e. DT1, DT2, DT3) between targets of two rules.*

Proof. Let us assume that there exists a hierarchical dependency between targets T1, T2 of two rules R1, R2. Again, let R1 and R2 be the only rules in the policy (thus, the size of BV vectors will be 2). We will show, that this dependency will be undoubtedly reflected by $\overrightarrow{BV}_T$ or $\overleftarrow{BV}_T$ vectors in procedure ConflictDetect of the main detection phase.

First, suppose that T1 and T2 are literally the same (T1=T2, thus, we will denote them simply as T). The proof that the IMCD algorithm will discover such a dependency is quite similar to the same dependency between subjects. Therefore, we simplify its description. According to line 7 of Alg. 1 $POG_T(T).BV[1] \leftarrow 1$ and $POG_T(T).BV[2] \leftarrow 1$, as T appears in both $R1$ and $R2$. Thus, $POG_T(T).\overrightarrow{BV} = [11]$ (from line 19) and $POG_T(T).\overleftarrow{BV} = [11]$

(line 26). When procedure ConflictDetect (Alg. 3) processes rule R1 it gets $\overrightarrow{BV}_T \leftarrow POG_T(T).BV = [11]$, as well as $\overleftarrow{BV}_T \leftarrow POG_T(T).BV = [11]$ which will further decide about the content of *PotentialConflictSet* (line 18). Thus, the knowledge about hierarchical dependency DT1 between targets of rule R1 and R2 will be preserved.

Next, let us consider dependency DT2 between targets T1 and T2. The proof will be almost the same as in corresponding parts of dependency DT2 between subjects proof. The main difference is that propagation of BV in POG_T creates two vectors $\overrightarrow{BV}$ for the descending propagation (the same as in POG_S), and $\overleftarrow{BV}$ for the ascending propagation.

1) Suppose that T2 is directly descendant from T1 ($T1 \rightarrow T2$), thus $POG_T(T1).BV[1] \leftarrow 1$ and $POG_T(T2).BV[2] \leftarrow 1$ (Alg. 1 line 7), $POG_T(T1).\overrightarrow{BV} = [10]$ and $POG_T(T2).\overrightarrow{BV} = [11]$ (line 19), and also $POG_T(T1).\overleftarrow{BV} = [11]$ and $POG_T(T2).\overleftarrow{BV} = [01]$ (line 26). When the main detection phase processes rule R1 (Alg. 3 line 5), according to line 13, finds $\overleftarrow{BV}_T \leftarrow POG_T(T1).\overleftarrow{BV} = [11]$. Since both, the first and the second bit of $\overleftarrow{BV}_T$ is set, IMCD realizes the dependency of targets of the first and the second rule (T1 and T2).

2) Suppose that T1 is directly descendant from T2 ($T2 \rightarrow T1$). Still $POG_T(T1).BV$=[10] and $POG_T(T2).BV$=[01], as before. The propagation gives $POG_T(T1).\overrightarrow{BV} = [11]$ and $POG_T(T2).\overleftarrow{BV} = [11]$. When the main detection phase processes rule R1, then $\overrightarrow{BV}_T \leftarrow POG_T(T1).\overrightarrow{BV} = [11]$ and $\overleftarrow{BV}_T \leftarrow POG_T(T1).\overleftarrow{BV} = [10]$. Again, the first and the second bits of $\overrightarrow{BV}_T$ are set, so IMCD realizes the dependency of targets of the first and the second rule (T1 and T2).

3) Suppose that $T2$ is *indirectly* descendant from $T1$. Then, according to lines 16–22 of Alg. 1, every direct descendant Tx of $T1$ will sum its $POG_T(Tx).BV$ and $POG_T(Tx).\overrightarrow{BV}$ with $POG_T(T1).\overrightarrow{BV}$=[10], and propagate the result down to its descendants. And every descendant will do the same with its own $\overrightarrow{BV}$ until $T2$. Finally, in $POG_T(T2).\overrightarrow{BV}$ the two bits will be set – the one corresponding to rule $R1$ (propagated initially from $POG_T(T1).\overrightarrow{BV}$) and the second corresponding to rule $R2$ (being already set in original $POG_T(T2).BV$). Moreover, the ascending propagation (lines 23–29) causes every direct ascendant Ty of $T2$ to sum its $POG_T(Ty).BV$ and now $POG_T(Ty).\overleftarrow{BV}$ with $POG_T(T2).\overleftarrow{BV}$=[01], and propagate the result up to its ascendants. And every ascendant will do the same with its own $\overleftarrow{BV}$ until $T1$. In consequence, $T1$ will finally get $POG_T(T1).\overleftarrow{BV}$=[11], i.e. the bits corresponding to both rules $R1$ and $R2$ will be set. The dependency between T1 and T2 will be found in $POG_T(T2).\overrightarrow{BV}$ (when looking from rule $R2$) and $POG_T(T1).\overleftarrow{BV}$ (when looking from rule $R1$). One can easily see that the same is true if $T1$ is indirectly descendant from $T2$.

We have shown in the three above points that the IMCD algorithm will detect dependency DT2 between targets T1 and T2.

Finally, assume hierarchical dependency DT3 (an inheritance cycle) between

T1 and T2. This proof will also be very similar to the proof of Lemma 1.

Suppose that T1, T2, Tx and Ty form an inheritance cycle in $\overline{POG_T}$ graph (Fig. 4). According to line 7 of Alg. 1 $POG_T(T1).BV[1] \leftarrow 1$ and $POG_T(T2).BV[2] \leftarrow 1$. In line 35 of Alg. 1 all the nodes in the cycle will finally have the same $\overrightarrow{BV}$, i.e. the sum of all $\overrightarrow{BV}s$ from these nodes, which results in $commonBV_T = [11]$. The main detection phase (Alg. 3) will set $\overrightarrow{BV}_T \leftarrow POG_T(T1).\overrightarrow{BV} = [11]$ for R1 (line 12). Because the first and the second bit of $\overrightarrow{BV}_T$ are set, IMCD sees the dependency between targets T1 and $T2$. Thus, also type DT3 of hierarchical dependencies between targets will be found.

We have shown that the IMCD algorithm will inevitably detect all dependencies (DT1, DT2, DT3) between targets T1 and T2 of rules $R1$ and $R2$. Q.e.d.

Having proven the above Lemmas, we will now use them to prove Theorem 1. To be precise, we will prove that the IMCD algorithm is able to include any two rules (R1, R2) in a *potential conflict set*, when at the same time their subjects are hierarchically dependent and their targets are hierarchically dependent.

Proof of Theorem 1. We will prove that our algorithm can detect an existing hierarchical dependency between subjects and targets of two rules. Again, let R1 and R2 be the only rules in the policy. Remark that a detection of a potential conflict between two rules will go through two rounds (lines 5–7 of Alg. 3), the first round for R1, and the second for R2. In many cases any one of those round will detect the conflict, but in few cases, shown further, only one of those rounds will do that correctly. Therefore, IMCD always needs to follow all the rounds (through all the rules, each one its own turn).

1. Lets start from the case that the existing dependency between S1 and S2 is of type DT1. From Lemma 1, this dependency will be detected. Then, any possible dependency between T1 and T2 will be detected from Lemma 2. In this case IMCD will detect the conflict in each round.
2. Next, assume that these subjects are in DT2 dependency and targets are in any possible dependency. This case is the most difficult, since IMCD may find the dependency between both rules only in one of the considered two rounds. Suppose, for example, that the dependency between subjects is $S1 \rightarrow S2$. When IMCD starts processing rule $R1$, then it will not discover a potential conflict (from Lemma 1 $BV_S = [10]$ and regardless of values of $\overrightarrow{BV}_T$ and $\overleftarrow{BV}_T$, the final vector $BV = [10]$ will not show any potential conflict yet.). However, in the other round, IMCD starts processing rule $R2$, and then $BV_S = [11]$. Now, (from Lemma 2) whatever dependency between targets will be, one of vectors $\overrightarrow{BV}_T$, $\overleftarrow{BV}_T$ will have value [11]. Concluding, from the final vector $BV = [11]$, IMCD will now know that both rules are in a potential conflict.

   Of course, a dependency between subjects can be $S1 \leftarrow S2$. Here, IMCD will discover the potential conflict for rule $R1$ ($BV_S = [11]$), but not for rule $R2$ (as $BV_S = [01]$).

3. Finally, consider dependency DT3 between subjects and any possible dependency between targets. From Lemmas 1 and 2 these dependencies will be detected in each round.

Q.e.d.

Theorem 2. *Algorithm IMCD discovers all pairs of conflicting rules in a security policy.*

Proof. From Theorem 1, all actually conflicting pairs of policy rules will be detected by the IMCD algorithm. For all those pairs, we will prove that if actions of both rules have common parts and conditions also have common parts, and decisions are contrary (i.e. a modality conflict occurs), the pair is included in the *ConflictSet* (i.e. finally detected).

First, in line 4 of Alg. 2, the IMCD algorithm checks the decisions of both rules. Next, the algorithm checks if both rules have common parts in an action parameter and in a condition parameter (line 6). For this purpose, it uses the naive algorithm. Since actions are represented by sets of operations, then the comparison gives intersection of the two sets. Comparing the conditions is more complicated, because there are composed of pairs of a condition type and a value. The algorithm must compare all conditions types separately. Note that if a given condition type does not exist in a rule, it means that any value is possible.

As a result of the above comparison, IMCD will include all pairs of rules matching the definition of a modality conflict in the *ConflictSet*. Thus, we have proven that the IMCD algorithm is able to detect all pairs of conflicting rules. Q.e.d.

4 Conclusions

In this work, we have discussed the problem of security policy verification, namely the discovery of modality conflicts in SOA-compliant security policies and presented a novel solution to this problem. Compared to the existing similar approaches, the proposed algorithm is general enough to handle any policy rules and has low time complexity. Along with the presentation of its algorithm, the solution has been formally proven correct.

The proposed solution has been implemented in the ORCA security policy framework [8]. At present, we have started experimental verification of its efficiency. This could give a better insight on the practical applicability of our proposal.

We are also working on continuous improvement of the IMCD algorithm. One of possible directions is the simplification of the policy principals representation. That could decrease the overhead of the preparation phase of the IMCD algorithm.

Acknowledgment. The research presented in this paper was partially supported by the European Union in the scope of the European Regional Development Fund program no. POIG.01.03.01-00-008/08.

References

1. Lupu, E., Sloman, M.: Conflicts in policy-based distributed systems management. IEEE Transactions on Software Engineering 25, 852–869 (1999)
2. Abassi, R., Fatmi, S.G.E.: Dealing with multi security policies in communication networks. In: 5th International Conference on Networking and Services, pp. 282–287 (April 2009)
3. Al-Shaer, E., Hamed, H.: Modeling and management of firewall policies. IEEE Transactions on Network and Service Management 1, 2–10 (2004)
4. Baboescu, F., Varghese, G.: Fast and scalable conflict detection for packet classifiers. In: 10th IEEE International Conference on Network Protocols, pp. 270–279. IEEE Comput. Soc., Los Alamitos (2002)
5. Craven, R., Lobo, J., Lupu, E., Russo, A., Sloman, M., Bandara, A.: A Formal Framework for Policy Analysis (2008)
6. Brodecki, B., Sasak, P., Szychowiak, M.: Security policy definition framework for SOA-based systems. In: Vossen, G., Long, D.D.E., Yu, J.X.Y. (eds.) WISE 2009. LNCS, vol. 5802, pp. 589–596. Springer, Heidelberg (2009)
7. Moffett, J.D., Sloman, M.S.: Policy conflict analysis in distributed system management. Journal of Organizational Computing 4, 1–22 (1994)
8. Brodecki, B., Szychowiak, M.: Conflict discovery algorithns used in ORCA. Technical Report TR-ITSOA-OB8-4-PR-11-03, Institute of Computing Science, Pozna University of Technology (2011)

FPGA Implementation of Variable-Precision Floating-Point Arithmetic

Yuanwu Lei, Yong Dou, Song Guo, and Jie Zhou

Department of Computer Science,
National University of Defence Technology,
Changsha, P.R. China, 410073
{yuanwulei,yongdou,songguo,zhoujie}@nudt.edu.cn

Abstract. This paper explores the capability of FPGA solutions to accelerate scientific applications with variable-precision floating-point (VP) arithmetic. First, we present a special-purpose Very Large Instruction Word (VLIW) architecture for VP arithmetic (VV-Processor) on FPGA, which uses unified hardware structure to implement various VP algebraic and transcendental functions. We take exponential and trigonometric functions (sine and cosine) as examples to illustrate the design of VP elementary algorithms in VV-Processor, where the optimal configuration is discussed in details in order to achieve minimum execution time. Finally, we create a prototype of VV-Processor unit and Boost Accelerator based-on VV-Processor into a Xilinx Virtex-6 XC6VLX760-2FF1760 FPGA chip. The experimental results show that our design, based on FPGA running at 253 MHz, outperforms the approach of a software-based library running on an Intel Core i3 530 CPU at 2.93GHz by a factor of 5-37X. Compared to the previous work, our design has higher performance and more flexibility to implement other VP elementary functions.

Keywords: Variable-precision floating-point (VP) arithmetic, Very Long Instruction Word (VLIW), elementary function, FPGA.

1 Introduction

Many scientific and engineering applications require efficient variable-precision floating-point (VP) arithmetic [1]. These applications range from mathematical computations to large-scale physical simulations, such as computational geometry, fluid dynamics, and climate modeling. It is extremely important to provide accurate results for the numerical sensitive calculations in these applications.

Most are accomplished using software approach, such as GNU Multiple-Precision library (GMP) [13], Multiple Precision Floating-Point Reliable library (MPFR) [11], and so on. The main drawback of software approach is speed. Compared with 64-bit floating-point arithmetic, software approaches are at least one order of magnitude slower for quadruple precision arithmetic and 40X slower for octuple precision arithmetic [12]. For the higher precision arithmetic, the computational time increases roughly quadratically with the precision.

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 127–141, 2011.

Some hardware designs attempt to overcome the speed limit of software solutions, such as CADAC [6], DRAFT [5], CASCADE [4], VPIAP [19], HPRCs [9], and HRAC [21]. Above processors were designed to perform basic VP arithmetic operations. Some studies focused on the special hardware for VP elementary functions such as logarithm, exponential, trigonometric [14], according to the properties of these functions.

Recently, Field-programmable gate array (FPGA) chips, which operate at the bit level and serve as custom hardwares for the different computation precision, could potentially implement high precision scientific applications and provide significantly higher performance than a general-purpose CPU [9][8][22]. However, several challenges still exist. First, the complex scientific computations require well implementation of elementary functions on FPGAs, such as logarithm, exponential, trigonometric, etc. The computation complexity of these functions is $O(n^{1/2}M(n))$[7], where $M(n)$ refers to the complexity of multiplication operation and n refers to the precision of result. The consumption of hardware resources increases quadratically relative to the precision with pipeline technology. Therefore, it is intractable to implement all of these elementary function units on the same chip. Second, the use frequency of elementary functions are much lower than the basic operations, so it will reduce the utilization of FPGA to cost much resources to implement all available elementary functions. Moreover, the latency of VP elementary function is long because higher internal working precision is required to obtain an accurate result.

In this paper, we present a special-purpose Very Large Instruction Word (VLIW) processor for VP arithmetic (VV-Processor), which uses the unified hardware to implement various VP algebraic and transcendental functions. Performance is improved by using explicitly parallel technology of VLIW and dynamically varying the precision of intermediate computation. Then, we take exponential and trigonometric functions (sine and cosine) as examples to illustrate the design of VP elementary functions in VV-Processor in details. Finally, we create a prototype of VV-Processor unit and Boost accelerator based-on VV-Processor into a Xilinx Virtex-6 XC6VLX760-2FF1760 FPGA chip.

2 Background

2.1 Range Reduction

The first step for the evaluation of elementary function f at x is to reduce the argument range, called *range reduction*, to improve computational efficiency. It is usually divided into three steps.

- *Range reduction*: transform x into x' based on the property of f like additivity, symmetry, and periodicity.
- *Evaluation*: evaluate f at x'.
- *Reconstruction*: compute $f(x)$ from $f(x')$ using a functional identity, and normalize it to the specific format.

2.2 Methods for Elementary Function

The main methods, used to evaluate high-precision elementary functions in hardware, are digit-recurrence, Newton's algorithm, and polynomial approximation. The two most well-known digit-recurrence algorithms are SRT algorithm [18] and CORDIC algorithm [15] [23], which are linearly converge and a fixed number bits of the result is obtained in each iteration. So the latency is very long to obtain a high-precision result. This is a serious shortcoming for high-precision computations.

Both Newton's algorithms and polynomial approximations are multiplicative approaches. Recently, computing-oriented FPGAs include plenty of embedded multipliers and RAM blocks. Multiplicative approaches allow one to make the best use of these available resources.

Newton's algorithm [3] can be used to compute functional inverse functions, such as reciprocal, division, and square root. A table lookup, storing the approximate value, is always followed to reduce the number of Newton's iteration. This method typically has quadratic convergence, which results in low latency for high-precision computations.

Polynomial approximation is another alternative to approximate elementary functions (exponential, logarithm, trigonometric functions). The degree of polynomial is proportional to the precision of result. Therefore, many multiplications and additions must be performed for high-precision computation. In practice, the range reduction technology is used to reduce the degree of polynomial.

2.3 Custom VLIW Architecture

Both Newton's method and polynomial approximation are combined with addition and multiplication operations and the elementary functions are constructed by basic arithmetic operations, so we can implement these elementary functions with unified hardware. However, data dependences between basic arithmetic operations are varied between different elementary functions.

Very Long Instruction Word (VLIW) [10] [16] is an effective approach to achieve high levels of instruction level parallelism (ILP) by executing long instruction words composed of multiple heterogeneous operations. This is a type of multiple instruction multiple data (MIMD) processor. We propose a custom VLIW architecture for VP arithmetic, which has the following advantages:

- The unified hardware, composed of basic VP arithmetic units, is used to evaluate a variety of VP elementary functions and data dependences between basic arithmetic operations are maintained with VLIW instruction.
- The performance is improved through explicitly parallel technology of VLIW and more available ILP can be exploited with loop unrolling technology.
- This custom VLIW processor achieves high scalability. Under the control of specific VLIW instructions, the basic VP arithmetic units are combined to a special-purpose hardware for VP elementary function which turn out to be much more efficient than a processor-based implementation.

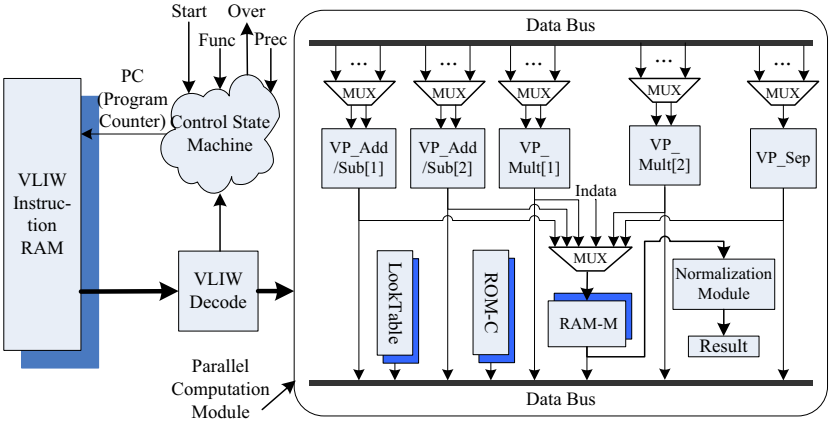


Fig. 1. Block diagram of the hardware structure of VV-Processor

3 Implementation of VV-Processor

3.1 Variable-Precision Floating-Point Format

The format of VP numbers is similar to the one presented in [20]. An VP number x consists of a 16-bit exponent field represented with a bias of 32,767 (BE_x, the real value of exponent is E_x=BE_x-32,767), a sign bit (S_x), a 7-bit mantissa length field (P_x), and a variable-precision mantissa field (M_x) that consists of P_x 64-bit words ($M_x[0]$ - $M_x[P_x\text{-}1]$, and $M_x[0]$ is the highest word). The value of the normalized mantissa is between 0.5 and 1, i.e., $0.5 \leq M_x < 1$. The precision of x is 64*P_x bits and the value of x is:

$$x = (-1)^{S_x} \cdot M_x \cdot 2^{BE_x - 32767} = (-1)^{S_x} \cdot M_x \cdot 2^{E_x}.$$

3.2 Hardware Organization

As shown in Fig. 1, VV-Processor is mainly composed of a parallel computation module, a control state machine module, a VLIW instruction RAM, and a VLIW decode module. The evaluation of all arithmetic functions are accomplished by the parallel computation module through explicitly parallel technology. This module consists of multiple custom basic VP arithmetic units, three on-chip memory modules, and control logic.

The basic VP arithmetic units include 2 VP multiplication (VP_Mult) units, 2 VP addition/subtraction (VP_Add/Sub) units, and a VP separator (VP_Sep) unit. As illustrated in [17], this configuration can obtain an better tradeoff between the performance and resource utilization. The VP_Mult unit performs VP multiplication operation (C=A×B), i.e., $C=(S_A \otimes S_B)*(M_A*M_B)*2^{E_A+E_B}$, where the mantissa lengths of A, B, C are m, n, r, respectively. In the multiplication of m word (64-bit) and n word mantissa, we use 4 64×64 fixed-point

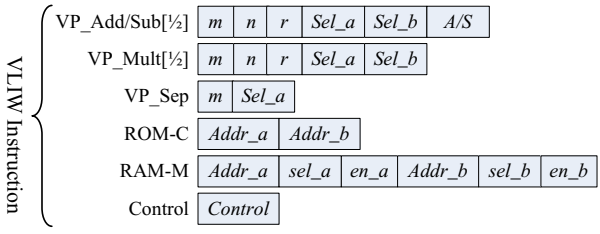


Fig. 2. The Field of custom VLIW instruction

multipliers in parallel to generate and accumulate $m*n$ partial products. The execution cycle is about $T_{\text{VP\_Mult}} = \frac{m*n}{4}+m+r+8$. The VP_Sep unit performs mod one operation, which separates an VP number A into an integer I and a fraction F, satisfied $A=I+F$, and $|F| < 1$. The VP_Add/Sub unit performs VP operation $C=A\pm B$. If it is assumed that $A\geq B$, then $C=2^{E_A}(S_A{*}M_A\pm S_B{*}M_B{*}2^{E_B-E_A})$. The execution time of VP_Add/Sub unit is about $n+r$.

Three on-chip memory modules are Lookup-Table, ROM-C, and RAM-M. The Lookup-Table, a 1024×9 ROM with a single port, stores the initial approximation for some functions. For example, it stores the initial value of reciprocal for the division operation and the approximation of reciprocal square root for the square root operation. ROM-C, a 8192×64 ROM with two read ports, is used to store constants. RAM-M, a 512×64 RAM with two ports, is used to buffer middle and final results. The port width of these memory modules is 64 bits and the maximum mantissa length is 63 64-bit words (4032 bits). A VP number is stored in 64 consecutive 64-bit words and the lowest word contains the exponent field, sign bit, and mantissa length field.

VLIW instruction RAM (VLIW-RAM) is a 2048×120 RAM. Each word is a VLIW instruction, as shown in Fig. 2, which consists of several fields including the data selecting field (*sel_a*, *sel_b*) and precision set field (m, n, r) for each basic arithmetic unit, write enable field (*en_a*, *en_b*), address field (*Addr_a*, *Addr_b*) for each on-chip memory module, control field, and so on. The decode module forms these fields from the VLIW instruction.

Control state machine module controls the run of VV-Processor. This module changes the value of program counter (PC) to access different VLIW instructions from VLIW-RAM to implement the corresponding VP elementary functions.

VV-Processor works through the following stages:

Stage 1, Start-up: After receiving the *Start* signal, the initial address to access VLIW-RAM is fixed according to the type of function (*Func*). At the same time, the initial data (*Indata*) is written into RAM-M.

Stage 2, Calculation: The VLIW instruction is read, the execution of computation module is controlled, and the value of PC is changed according to the control field in VLIW instruction.

Stage 3, Normalization: The *Over* signal is generated after the calculation of given function is done.

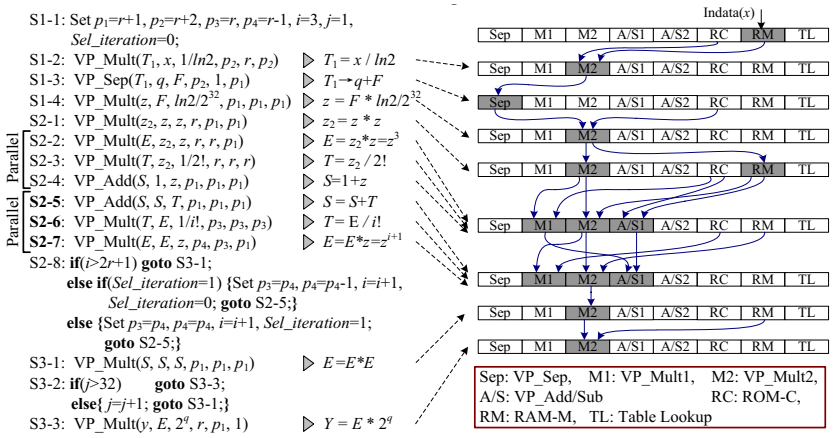


Fig. 3. VP exponential function algorithm and VLIW instructions

4 Variable-Precision Elementary Function Algorithm

For each VP elementary function, we need to design corresponding VLIW instructions and dynamically specify the precision of the basic operations to yield the desired result and achieve high performance. First, the elementary function is decomposed into a series of basic VP arithmetic operations, which can be executed by the basic arithmetic units. Then, these basic operations are mapped into the custom VLIW instructions according to the data dependent between them. Finally, the corresponding VLIW instructions are executed though the explicitly parallel technology of multiple basic arithmetic units.

In this section, we take exponential and trigonometric functions as examples to illustrate the design of VP elementary functions in VV-Processor in details. The division and square root are implemented with table lookup and Newton's method. The logarithm algorithm uses an iterative technique that uses table lookup and polynomial approximation as described in [14].

4.1 VP Exponential Function Algorithm

Algorithm: As shown in Fig. 3, the VP exponential algorithm (y=e^x) is based on the Taylor series and performed through three stages as follows.

Stage 1, Range reduction: First, the argument x is reduced into interval [0, ln2). We find s and an integer q, satisfied that x=q*ln2+s and $0 \leq s < \ln 2$. Then $y = e^x = e^s * 2^q$. Thus, the evaluation of the exponential function on the real field is transferred into that on interval [0, ln2). To derive the value of s and q, x is multiplied by 1/ln2, and the integer part of product is q. The fraction part of product is multiplied by ln2 to gain the value of s.

To reduce the degree of Taylor series, we repeat the doubling formula ($e^{2x} = (e^x)^2$) to reduce the argument further. Given $z = s/2^m$, then $e^s = (e^z)^{2^m}$ and

$|z| < 2^{-m}$. This is better since the power series converges more quickly for z. The cost is that m squaring are required to reconstruct the final result from e^z. So, there is some compromise in choosing the value of m, as analyzed following.

Stage 2, Evaluation: e^z can be evaluated using the Taylor series approximation approach:

$e^z = \sum_{n=0}^{i} \frac{z^n}{n!} = 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \frac{z^4}{4!} + \cdots$

Stage 3, Reconstruction: The value of e^s is calculated m times using the doubling formula starting from e^z.

Analysis: The Taylor series approach in Stage 2 is a kind of polynomial ($F(i)$=$F(i$-1)+a_i*x^i). As shown in Fig. 3, S2-5 implements addition operations ($F(i$-1)+a_i*x^i) in i^{th} iteration, and S2-6 implements multiplication operations ($a_{i+1}x^{i+1}$) in $(i$+1)th iteration, and S2-7 implements multiplication operations (x^{i+2}) in $(i$+2)th iteration of the polynomial approximation approach. There is no data dependent between S2-5, S2-6, and S2-7, so they can be integrated into one VLIW instruction, and executed simultaneously. Therefore, the critical path in Stage 2 includes only one VP multiplication or addition/subtraction.

Since at most one bit of accuracy will lose in each multiplication operation [8], the precision of result in Stage 2 should be greater than $64r$+m. The error in Stage 2 comes from two sources: approximation error (ε_a) and rounding error (ε_r). The $|z| < 2^{-m}$ and the ignored terms in the Taylor series produce the approximation error

$\varepsilon_a = \sum_{j>i} z^j/j! < z^{i+1} < 2^{-m(i+1)} \leq 2^{-(64r+m)}$.

Thus, we need to calculate the front $\lceil 64r/m \rceil$ terms of the Taylor series. Rounding error occurs when calculating the Taylor series. If we use $64(r$+1) bits as the internal working precision and the number of operations is smaller than 2^m, the rounding error ε_r will be smaller than $2^{-(64r+m)}$.

Optimal Configuration: The execution time are mainly consumed in Stage 2 and Stage 3. We can choose an optimized value of m for different precision to balance the execution in Stage 2 and Stage 3 and achieve the minimum execution cycle. As analysis above, $\lceil 64r/m \rceil$ multiplication operations and m multiplication operations are needed in Stage 2 and Stage 3, respectively. Therefore, the execution time in Stage 2 is increasing with m, but that in Stage 3 is reduced with m. The precision of multiplication in Stage 2 are varied from $64(r$+1) to 64, as illustrated in Fig .3, so the execution cycle in this stage is about:

$C_2 = \frac{64r}{m(r+1)} \sum_{n=1}^{r+1} (\frac{n^2}{4} + 2n + 8) = \frac{8r(2r^2+31r+246)}{3m}$.

In Stage 3, the precision of multiplication operations are all $64(r$+1), so the execution cycle in this stages is about:

$C_3 = 0.25(r^2 + 10r + 41)m$.

Thus, total execution cycle in critical path is about $C_2 + C_3$ and the condition to achieve the minimal execution time is:

$m = \sqrt{\frac{32r(2r^2+31r+246)}{3(r^2+10r+41)}}$.

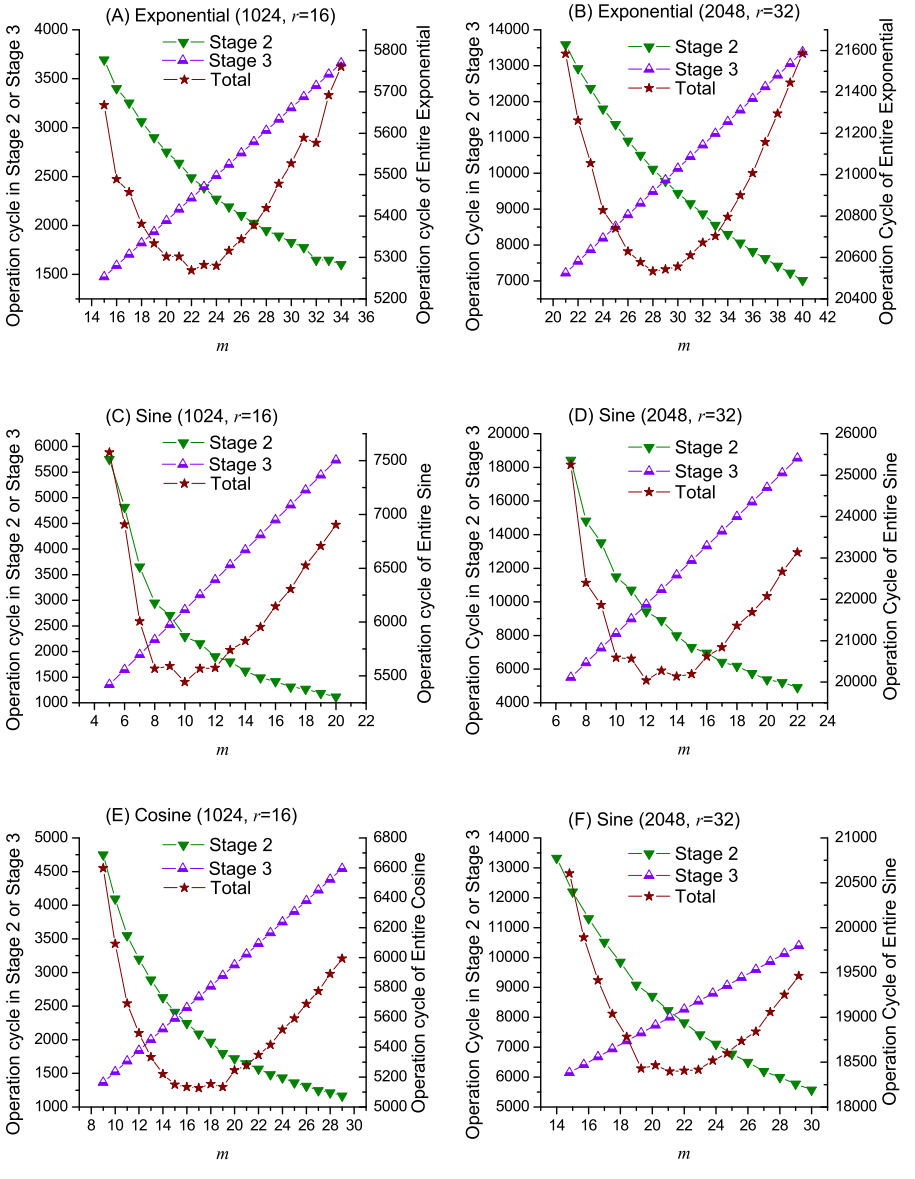


Fig. 4. The execution cycle of VP exponential, sine, and cosine function for different value of m

From the above equation, we predict that $m = 21.64$ and $m = 28.48$ are the best setting for the precision of 1024-bit and 2048-bit, respectively. The actual results in Fig .4 (A) and (B) are $m = 22$ and $m = 28$, that verified the prediction.

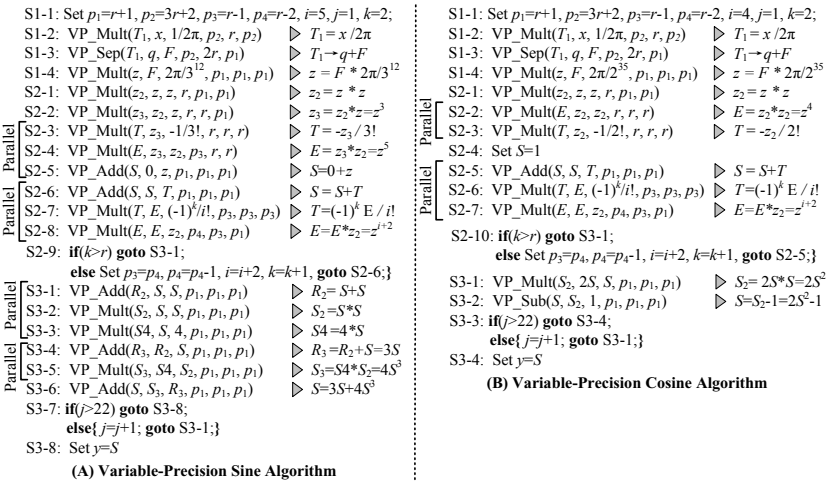


Fig. 5. VP sine and cosine algorithms

4.2 VP Trigonometric Function Algorithm

Algorithm: As shown in Fig. 5(A) and (B), the approach based on Taylor series to calculate VP sine function (y=sin(x)) and cosine function (y=cos(x)) works as following:

Stage1, Range reduction: First, the argument x is reduced into interval $[0, 2\pi)$. We find s and an integer q, satisfied that x=q*2π+s, with the similar scheme of computation in Stage 1 in exponential function. Then y= sin(q*2π+s)=sin(s) and y= cos(q*2π+s)=cos(s). Tripling formula ($\sin(3x) = 3\sin(x) - 4\sin^3(x)$) and double formula ($\cos(2x) = 2\cos^2(x) - 1$) are used to reduce the argument further for VP sine and cosine function, respectively.

Stage2, Evaluation: The value of sin(z) and cos(z) can be evaluated using Taylor series approximation approach:

$$\sin(z) = \sum_{n=0}^{i} \frac{(-1)^n z^{2n+1}}{(2n+1)!} = z - \frac{z^3}{3!} + \frac{z^5}{5!} - \frac{z^7}{7!} + \cdots.$$

$$\cos(z) = \sum_{n=0}^{i} \frac{(-1)^n z^{2n}}{(2n)!} = 1 - \frac{z^2}{2!} + \frac{z^4}{4!} - \frac{z^6}{6!} + \cdots.$$

Stage3, Reconstruction: The value of sin(s) is calculated using tripling formula m_s times starting from sin(z), and the value of cos(s) is calculated using double formula m_c times starting from cos(z).

Optimal Configuration: Similar to the analysis of VP exponential function, we can choose an optimized value of m_s and m_c for different precision to balance the execution in Stage 2 and Stage 3 of VP sine and cosine functions and achieve the minimum execution cycle.

For VP sine function, the execution cycles in Stage 2 and Stage 3 are about: $C_{s2} = \frac{1}{2}(\frac{64r\log 2}{m\log 3 - \log(2\pi)} - 3)\sum_{n=1}^{r+1}(\frac{n^2}{4} + 2n + 8)$,

Table 1. Synthesis results

Type	Slice LUT	DSP48E1	BRAM	Freq(MHz)
VP_Mult	4095	48	0	262.03
VP_Add	2491	0	0	296.57
VP_Sep	986	0	0	287.55
VV-Processor	13744(2%)	96(11%)	43(3%)	253.45
Boost_Acc	42101(8%)	288(33%)	172(12%)	242.94

$C_{s3} = (0.5r^2 + 7r + 20)m$.

Thus, the condition to achieve minimal execution time in VP sine is:

$m_s = \sqrt{\frac{32r(2r^2+31r+246)\log 2}{24(0.5r^2+7r+20)\log 3}} - \frac{\log(2\pi)}{\log 3}$.

For VP cosine function, the execution cycles in Stage 2 and Stage 3 are about:

$C_{c2} = \frac{1}{2}(\frac{64r}{m-3} - 2)\sum_{n=1}^{r+1}(\frac{n^2}{4} + 2n + 8)$,

$C_{c3} = 0.25(r^2 + 18r + 41)m$.

Thus, the condition to achieve minimal execution time in VP cosine is:

$m_c = \sqrt{\frac{16r(2r^2+31r+246)}{3(r^2+18r+41)}} + 3$.

From the above equation, we predict that $m = 9.7$ and $m = 12.4$ are the best setting for VP sine function and $m = 16.5$ and $m = 21.4$ are the best setting for VP cosine function. The actual results in Fig .4 (C-F) verify this prediction.

5 Experiments Result

We implemented the proposed hardware design on FPGAs. All modules are coded in Verilog, and synthesized with Xilinx ISE 12.3 using Virtex-6 XC6VLX760-2FF1760 FPGA chip. As a base for comparison, we applied the MPFR library, MPFR3.0.0, developed by Hanrot et al., to measure the results accuracy and delay time. This library is quite efficient and accuracy compared to other multiple-precision library [11]. The software platform includes a host PC with 2.93GHz Intel Core i3 530 CPU and 4GB DDR3 1333MHz Memory.

5.1 Resource Utilization

Table 1 shows details of the FPGA synthesis results for basic VP arithmetic units and VV-Processor unit. The DSP48E1 blocks, used to build 64-bit fixed-point multiplication module in VP_Mult, are the most constrained resource. Since the VP_Mult module is a key unit in VV-Processor, we equip four (p=4) parallel 64-bit fixed-point multipliers to obtain the best tradeoff between the performance and resource utilization in our design. One VV-Processor unit consumes 11% DSP48E available in XC6VLX760 FPGA chip. Therefore, a total of 9 VV-Processor modules can be integrated into it.

Table 2. Timing in microsecond comparison with MPFR library. The Inputs correspond to $x = \sqrt{3}$, $y = \sqrt{5}$.

Op	1024 bits			2048 bits		
	MPFR	Our	Speedup	MPFR	Our	Speedup
$x \pm y$	0.7	0.126	**5.6**	1.25	0.25	**5**
$x \times y$	12.9	0.41	**31.5**	32.18	1.30	**24.8**
x/y	18.6	1.95	**9.5**	64.1	5.05	**12.7**
$\sqrt{x}$	18.8	2.52	**7.5**	46.9	6.39	**7.3**
$\mathrm{Sin}(x)$	458	21.5	**21.3**	1766	79.2	**22.3**
$\mathrm{Cos}(x)$	405	20.3	**20.0**	1640	72.7	**22.6**
$\mathrm{Exp}(x)$	420	20.8	**20.2**	1515	81.2	**18.7**
$\mathrm{Ln}(x)$	579.7	15.7	**36.9**	1547	46.1	**33.6**

The local memory modules (on-chip memory), classified into distributed RAM and embedded 18Kbits block RAM (BRAM), are important in the implementation of VV-Processor. We used 29 and 14 block RAM to implement ROM-C (8192×64bit) and VLIW instruction RAM (2048×120bit), respectively. The more available storage resources in current FPGAs can be used to build a larger table lookup, which provide a rough approximation to the elementary function in range reduction scheme. Thus, the latency of evaluation is reduced further. We used distributed RAM to implement other small on-chip memory, such as RAM in basic VP arithmetic units. The distributed RAM elements are mapped into LUT slices, which is an advantage for the FPGA placement and layout phase, compared with fix-positioned embedded block RAMs. However, distributed RAMs consume more LUT resources. An VV-Processor requires about 2% of the slice LUT resources available in XC6VLX760-2FF1760.

5.2 Evaluation of VV-Processor

The accuracy of the proposed methods was tested by comparing to the accurate results produced by 4096 bits precision calculations using the MPFR library. For the elementary function, we designed corresponding variable-precision algorithms, in which the internal precision is carefully planed and guard words are used to guarantee accuracy of the result. The experiment results show that we can obtain results with correctly rounding for addition, multiplication, that ulp (unit in last place) error is not exceeding 0.5, and the error is smaller than 0.55 ulp for VP elementary functions.

Table 2 compares the performance of VV-Processor with the similar precision implementation of MPFR library [11], which is a quite efficient variable-precision binary floating-point library. The speedup factor for basic arithmetic operations (addition, multiplication, division, and square root) is between 5 and 31.5, and that for elementary functions (exponential, sine, cosine and logarithm) is between 18 and 36.9. In the logarithm algorithm, the reconstruction, which occupies half of execution time for others transcendental functions, is simple. Therefore, the

Table 3. Performance (cycle) comparison with VPIAP[19] and CORDIC[15]

Op	1024 bits			2048 bits		
	[19]/[15]	Our	Speedup	[19]/[15]	Our	Speedup
$x \pm y$	40	32	**1.25**	72	64	**1.13**
$x \times y$	284	104	**2.96**	1068	328	**3.33**
x/y	852	493	**1.73**	3220	1280	**2.52**
$\sqrt{x}$	1146	639	**1.79**	4314	1620	**2.66**
$\mathrm{Sin}(x)$	11K	5.3K	**2.08**	38K	19.6K	**1.94**
$\mathrm{Cos}(x)$	11K	5.0K	**2.2**	38K	18.0K	**2.11**
$\mathrm{Exp}(x)$	11K	5.1K	**2.16**	38K	20.0K	**1.90**
$\mathrm{Ln}(x)$	11K	3.8K	**2.89**	38K	11.4K	**3.33**

latency of VP logarithm function is smaller than that of others. However, more on-chip memory is required to store the approximation in lookup tables. The performance of FPGA implementation of VP applications is increase with the number of integrated VV-Processor units.

5.3 Performance Comparison with Related Work

In table 3, we compare the performance of our design to two existing design, VPIAP processor[19] and CORDIC processor[15]. The VPIAP processor was designed to perform the basic VP arithmetic operations. The performance of addition in our design is about the same. For multiplication operation, the speedup of 3X is obtained, since multiple fixed-point multipliers are used to generate and accumulate the partial products in parallel in the multiplication of M_A and M_B. For division and square root, Newton-Raphson iteration with table-based method is used. The speedup over 1.7X is obtained for division and square root.

The CORDIC processor, which was designed to perform VP transcendental functions. Since the approach based on multiplication operations has high level convergence and dynamically varies the precision of intermediate computation to reduce the latency further, our proposal outperforms the CORDIC processor by a factor of 1.8X-3.3X. Moreover, the CORDIC algorithm cannot guarantee low relative error [23]. It needs higher computation bandwidth, more storage to store the elementary rotation angle, and more iterations to obtain a desired result, when the result is close to zero.

In sum, comparison to the work in [19] and [15], our design can implement various VP algebraic and transcendental functions in the unified hardware.

5.4 Boost Accelerator

Implementation: We present a Boost Accelerator (Boost_Acc), which are used to evaluate the special functions of boost C++ library [2], as example to illustrate the performance of VV_Processor. As shown in Fig. 6(A), we implemented

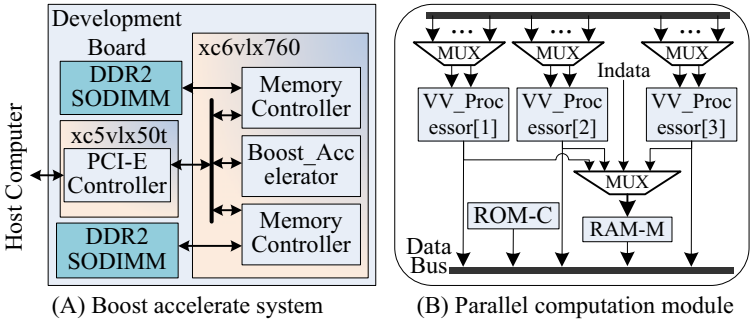


Fig. 6. Block diagram of Boost accelerator

Table 4. Performance comparison for Boost C++ library

Program	1024 bits			2048 bits		
	MPFR (s)	Boost-Acc (s)	speedup	MPFR (s)	Boost-Acc (s)	speedup
Ellint_rf	36.3	1.9	18.9	164.0	10.1	16.2
Ellint_rd	14.0	0.76	18.4	71.1	3.9	18.0
Ellint_1	36.3	1.9	18.9	164.0	10.1	16.2
Ellint_2	62.0	3.2	19.2	322.0	17.0	18.9
Bessel_ik	1957.0	107.6	18.2	19887.9	1176.8	16.9

the hardware system on a development board mainly consisting of two FPGAs (Virtex-5 XC5VLX50T-1FF1136 and Virtex-6 XC6VLX760-2FF1760) and two 2GB DDR2 DRAM modules. The XC6VLX760 chip is used to implement the our design. The XC5VLX50T chip provides a link between the XC6VLX760 and host PC via a 8-line PCI-Express bus with bandwidth of 570 MB/s. Each DDR2 Controller runs at 200MHz on 128bit data width and the peak I/O bandwidth can reach 6.4 GB/s. The running time of Boost accelerator includes computation time and the time for sending the operands to FPGA as well as receiving the results back to the host PC.

Similar to VV-Processor, the Boost Accelerator is a custom VLIW architecture, which is mainly composed of VLIW instruction RAM, control state machine, parallel computation module, and others logic. As shown in Fig. 6(B), the parallel computation module is equipped with three VV-Processor units.

Performance: We use Boost accelerator to implement two special functions, which are Elliptic integral and Bessel Functions. They are widely used in scientific and engineering applications. In Table 4, the programs (*ellint_rf*, *ellint_rd*, *ellint_1* and *ellint_2*) calculate the formula (1), which return the results of elliptic integral functions, respectively. The program (*Bessel_ik*) calculations the formula (2) and returns the results of $I_v(z)$ and $K_v(z)$ simultaneously by Temme's method. In this paper, we evaluate various special functions on the vector X, Y, Z, φ, K, and V where the size of these vectors are 1M and the elements

are randomly generated between 0 and 1. As depicted in Table 4, compared to MPFR library, this accelerator can achieve a speedup factor of 16X-19X.

$$\begin{cases} R_F(x,y,z) = 0.5\int_0^\infty (t+z)^{-0.5}[(t+x)(t+y)]^{0.5}dt \\ R_D(x,y,z) = 1.5\int_0^\infty (t+z)^{-1.5}[(t+x)(t+y)]^{0.5}dt \\ F(\varphi,k) = \int_0^\varphi (1-k^2\sin^2\theta)^{-0.5}d\theta \\ E(\varphi,k) = \int_0^\varphi \sqrt{1-k^2\sin^2\theta}d\theta \end{cases} \tag{1}$$

$$I_v(z) = (0.5\cdot z)^v \sum \nolimits_{k=0}^{\infty}(0.25\cdot z^2)^k/(k!\Gamma(v+k+1)) \tag{2}$$

6 Conclusion

We presented a custom processor for variable-precision floating-point arithmetic processor based on VLIW structure, that used the explicitly parallel technology of multiple basic arithmetic units to improve the performance. Several VP elementary function algorithms are designed in VV-Processor, where several tradeoffs are taken into consideration. The experimental results show one VV-Processor could achieve 5X-37X performance speedup compared with MPFR library. Moreover, our hardware design can achieve better performance than the related works.

Acknowledgements. This work was partially supported by NSFC (60833004) and 863 (2008AA01A201).

References

1. Bailey, D.H.: High-precision floating-point arithmetic in scientific computation. Computing in Science and Engineering 7(3), 54–61 (2005)
2. Boost C++ libraries, `http://live.boost.org/`
3. Brent, R.P., Zimmermann, P.: Modern Computer Arithmetic. Cambridge University Press, Cambridge (2010)
4. Carter, T.M.: Cascade: Hardware for high/variable precision arithmetic. In: Proceedings of the 9th Symposium on Computer Arithmetic, pp. 184–191 (1989)
5. Chiarulli, D.M., Ruaa, W.G., Buell, D.A.: Draft: A dynamically reconfigurable processor for integer arithmetic. In: Proceedings of the 7th Symposium on Computer Arithmetic, pp. 309–318 (1985)
6. Cohen, M.S., Hull, T.E., Hamarcher, V.C.: Cadac: A controlled-precision decimal arithmetic unit. IEEE Transactions on Computers C-32, 370–377 (1983)
7. Computational complexity of mathematical operations, `http://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations`
8. Dou, Y., Lei, Y., Wu, G.: FPGA accelerating double/quad-double high precision floating-point application for exascale computing. In: Proceedings of ICS 2010, pp. 325–336 (2010)
9. El-Araby, E., Gonzalez, I., El-Ghazawi, T.: Bringing high-performance reconfigurable computing to exact computations. In: Proceedings of FPL 2007, pp. 79–85 (August 2007)

10. Fisher, J.A.: Very Long Instruction Word architectures and the ELI-512. In: Proceedings of the 10th Annual International Symposium on Computer Architecture, pp. 140–150 (1983)
11. Fousse, L., Hanrot, G., Lefevre, V., Pelissier, P., Zimmermann, P.: MPFR: A multiple-precision binary floating-point library with correct rounding. Transactions on Mathematical Software 33(2), 1–15 (2007)
12. Fujimoto, J., Ishikawa, T., Perret-Gallix, D.: High precision numerical computations-a case for an happy design, ACPP IRG note, ACPP-N-1: KEK-CP-164 (May 2005)
13. GNU Multiple-Precision arithmetic library, `http://www.swox.com/gmp`
14. Hormigo, J., Villalba, J., Schulte, M.: A hardware algorithm for variable-precision logarithm. In: Proceedings of ASAP 2000, pp. 215–224 (July 2000)
15. Hormigo, J., Villalba, J., Zapata, E.L.: Cordic processor for variable-precision interval arithmetic. Journal of VLSI Signal Processing 37, 21–39 (2004)
16. Jones, A.K., Hoare, R., Kusic, D.: An FPGA-based VLIW Processor with Custom Hardware Execution. In: Proceedings of FPGA 2005, pp. 107–117 (2005)
17. Lei, Y., Dou, Y., Zhou, J., Wang, S.: VPFPAP: A special-purpose VLIW processor for variable-precision floating-point arithmetic. In: Proceedings of FPL 2011 (2011)
18. Parhi, K.K., Srinivas, H.R.: A fast radix-4 division algorithm and its architecture. IEEE Transactions on Computers 44(6), 826–831 (1995)
19. Schulte, M.J., Swartzlander Jr., E.E.: A family of variable-precision, interval arithmetic processors. IEEE Transactions on Computers 49(5), 387–397 (2000)
20. Schulte, M.J., Swartzlander Jr., E.E.: Hardware design and arithmetic algorithms for a variable-precision, interval arithmetic coprocessor. In: Proceedings of the 12th Symposium on Computer Arithmetic, pp. 222–228 (1995)
21. Tenca, A.F., Ercegovac, M.D.: A Variable Long-Precision Arithmetic Unit Design for Reconfigurable Coprocessor Architectures. In: Proceedings of FCCM 1998 (1998)
22. Underwood, K.: FPGAs vs. CPUs: trends in peak floating-point performance. In: Proceedings of FPGA 2004, pp. 171–180 (2004)
23. Zhou, J., Dou, Y., Lei, Y., Xu, J., Dong, Y.: Double precision hybrid-mode floating-point fpga cordic coprocessor. Proceedings of HPCC 2008, 182–189 (2008)

Optimization of *N*-Queens Solvers on Graphics Processors

Tao Zhang[1,2], Wei Shu[2], and Min-You Wu[1]

[1] Shanghai Jiao Tong University, Shanghai, China
{tao.zhang,mwu}@sjtu.edu.cn
[2] University of New Mexico, Albuquerque, USA
shu@ece.unm.edu

Abstract. While graphics processing units (GPUs) show high performance for problems with regular structures, they do not perform well for irregular tasks due to the mismatches between irregular problem structures and SIMD-like GPU architectures. In this paper, we explore software approaches for improving the performance of irregular parallel computation on graphics processors. We propose general approaches that can eliminate the branch divergence and allow runtime load balancing. We evaluate the optimization rules and approaches with the n-queens problem benchmark. The experimental results show that the proposed approaches can substantially improve the performance of irregular computation on GPUs. These general approaches could be easily applied to many other irregular problems to improve their performance.

Keywords: GPU, N-queens, Irregular, Divergence, Load Balancing.

1 Introduction

GPUs have been used in various computing areas like molecular dynamics, astrophysics simulation, life sciences, MRI reconstruction and so on, achieving more than 100x speedups over their CPU counterparts [1]. However, some irregular applications, like 3D-lbm and gafort, are not so suitable for executing on GPUs, showing poor speedups [2]. The programs of irregular applications generally contain complex control flow, which causes irregular memory access, branch divergence, and load imbalance. GPUs suffer branch divergence because they are composed of one or more SIMD-like streaming processors (named StreamMultiprocessors or SMs by Nvidia) which require their scalar pipelines to execute the same instructions together. Although general optimization rules and techniques are still more or less effective for solving irregular problems on GPUs, more insight and approaches developed specifically for irregular problems are imperative for further improvement of performance.

In this paper we focus on investigating and solving the performance issues found by implementing and analyzing different version of the n-queens problem benchmark on the Nvidia GTX480 GPU. The computation structures of the n-queens problem are highly irregular. Different sizes of the problem have different

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 142–156, 2011.

parallelism and thread granularity, which provide desired inputs for our study on memory access, branch divergence and load balance. Although our final n-queens kernel might be the fastest n-queens solver on GPUs, we do not aim only at optimizing this benchmark but instead at devising general rules and approaches for all irregular applications. Besides the proposed optimization approaches, we also present many useful analyses and interesting discussions, and give details on how to apply the proposed optimization approaches on other irregular applications. Overall, our major contributions are as follows:

- We propose an IRRC (Iteration-Related Reorganization of Computation) approach to reorganize the computation inside loop structures to reduce branch divergence on GPUs. The approach is simple to apply and incurs no extra overhead in execution time.
- We propose a distributed job pool approach and a monolithic job pool approach to balance the load at runtime. The two approaches are the first to address the load imbalance on GPUs at thread level. They are easy to use and need no extra memory.

2 Related Work

GPUs have already proved useful on regular problems like matrix multiplication and so on. Recently, much research has focused on solving irregular problems on GPUs [3,4,5,6,7,8] in order to exploit the power of GPUs on a wider range of applications. In all the issues found in previous work, branch divergence and load imbalance are the major issues that cause reduction of performance.

Branch divergence has long been a consideration in the design of GPU architectures [9,10] as well as GPU applications [2]. This issue can be addressed by hardware or software approaches. Fung et al. [10] proposed a hardware approach to form and schedule new warps dynamically every cycle. The basic idea is to group the threads taking the same code branches into the same warps so that the branch divergence is reduced. Zhang et al. [2] presented a sophisticated software method to eliminate thread divergence through runtime thread-data remapping. Their method is effective for those applications that expose data-branch patterns, but the remapping process incurs some overhead. Our approach is to adjust the computation inside the iterations, which is simple to use and introduces no overhead.

Load balancing on GPUs has become a recent topic because of the advent of more elegant synchronization hardware components such as atomics, and the need to exploit the power of GPUs on more complex graphics rendering or scientific computation. Previous approaches to load balancing work on two levels: block-level [3] and warp-level [4,5]. Block-level load balancing moves the work between hardware processors while warp-level balancing adjusts the work between warps (a warp is a group of a certain number of threads, often 32, that executes on a single streaming processor). The imbalance they try to address is caused by uneven distribution of aggregate jobs (jobs and new jobs generated during execution) among hardware processors. However, the n-queens problem doesn't

generate new jobs during execution. Instead the load imbalance is introduced from the variance of thread granularity within the same warp. This imbalance cannot be resolved using existing approaches, hence we propose new approaches to address this issue.

The n-queens problem is a typical constraint satisfaction problem. The ultimate goal of the n-queens problem is to find distinct solutions to place N queens on an $N \times N$ chess board. This problem has useful applications in a wide range of areas: parallel memory storage approaches, image processing, VLSI testing, traffic control, deadlock prevention, physical & chemical studies and networks [11]. The total solution cost for the n-queens problem increases exponentially with N [12]. As a result, it is placed in the NP (Non-Deterministic Polynomial) complexity class [13]. Currently the solution count for the n-queens problem for N larger than 26 is still unknown. The "QUEENS@TUD" project carried out by the TUD university found the solution count for the 26-queens problem with 9 months' computation effort on massively-parallel FPGA-based devices. This result was confirmed later by the Russian "MC#" project which solved the 26-queens problem on two supercomputers of the Top500 list [14]. The n-queens problem was also studied as an asychronous/irregular application in computer architecture literature. Shu et al. [15] proposed a high-level runtime support system (the P Kernel) to run loose asynchronous and asynchronous applications including the n-queens problem on an SIMD machine MasPar MP-1. Similar to Shu's work, Blas et al. [16] designed a low-level program coding methodology for the same purpose.

3 Design and Implementation of the Naive N-Queens Kernel

To solve the n-queens problem on parallel GPU cores, we can partition the problem into a number of disjoint sub-problems by placing k queens in the top k rows on a $N \times N$ board. Then each sub-problem corresponds to one valid placement. The symmetry feature of the n-queens problem is considered to reduce the computation task by around half [17]. To save memory space, each sub-problem is represented by three 32-bit unsigned integers that record the aggregate effect of all the rows above on three directions: vertical, left-diagonal and right-diagonal. Usually, the workload of partitioning is much smaller than that of processing the sub-problems, provided that k is relatively small. Therefore, the partitioning of the problem is carried out on the CPU, and the generated sub-problems are then processed on the GPU. In this work, we focus on the study of the workload on the GPU. In Nvidia's convention, the part of code invoked by a CPU and executed on a GPU is called the *kernel*. Figure 1 shows the pseudocode of the naive n-queens kernel.

The register variable "rowIndex" represents the index of the current row, with initial value 0. The "job_data" points to the memory space in global memory that stores the sub-problems. Each sub-problem occupies three 32-bit unsigned integers. The "results" points to the memory space in global memory to write

```
__global__ void nqueen_kernel_0(*job_data, *results, *work_space... )
{
   __register__  rowIndex, solution;

   each thread fetches a task from job_data into its array ROW[ ] in work_space;
   while(rowIndex >= 0) {
      if (no position to place new queen in ROW[rowIndex]) { rowIndex--; }
      else{
         finds a valid position P in ROW[rowIndex];
         places a queen at P in ROW[rowIndex] and mark the position as occupied;
         if (reaches last row) { solution++; }
         else{
            generates ROW[rowIndex+1] based on Row[rowIndex] and the position  P;
            rowIndex++;
         }
      }
   }

    reduction of the solutions of the threads within each block;
}
```

Fig. 1. Pseudo Code of the Naive N-queens Kernel

Table 1. Results of the Naive N-queens Kernel

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.40	1.46	11.03	78.51	664.13

back the total solution count. The "work_space" points to the memory space in global memory for threads to perform computations. Each GPU thread has its own portion in the "work_space", which can hold forty-four 32-bit unsigned integers. At the beginning of the kernel execution, each thread acquires a job (a sub-problem of the n-queens as described above) from the "job_data" array in global memory. Then threads enter a while loop to search for all valid solutions. The performance of the naive n-queens kernel on different N is shown in Table 1. The time in all the tables thereafter is the time for executing the kernels on the GPU, which does not include the time for the CPU to partition the n-queens problem into sub-problems.

4 Optimization

4.1 Optimal Usage of Memory

Enlarge the L1 Cache. In the naive n-queens kernel, the "job_data", "results", and "work_space" are all allocated in the global memory of the GPU. The "work_space" will be read and written many times by threads. For instance, there will be more than 4.76E+10 32-bit accesses (read or write) to the "work_space" during the computation for the 18-queens problem according to our analysis. We noticed that modern GPUs like the Nvidia GTX480 have shared memory and L1

Table 2. Results of the Naive N-queens Kernel with 48KB L1 Cache

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.37	1.26	9.54	67.94	575.54
Speedups	1.08	1.16	1.16	1.16	1.15

cache that share a configurable space. By default, the GTX480 is configured to have 48KB shared memory along with 16KB L1 cache in each SM. By configing to 48KB L1 cache with 16KB shared memory in each SM, the performance result is shown in Table 2. The speedups refer to the time ratios of the naive kernel with 16KB L1 cache (Table 1) versus the naive kernel with 48KB L1 cache. On average, the kernel runs 1.14x faster thanks to the larger L1 cache.

Utilize the Shared Memory. Registers and Shared memory are much faster than global memory because they are on-chip (around 4, 40, and 440 clock cycles per access respectively). Therefore we revised the naive n-queens kernel to move the "work_space" from global memory to shared memory. The GPU was changed back to have 48KB shared memory and 16KB L1 cache in each SM. The performance result is shown in Table 3. The speedups refer to the time ratios of the naive n-queens kernel with 48KB L1 cache (Table 2) versus this new kernel that utilizes the shared memory. On average, the kernel runs 4.49x faster because of the fast-access shared memory.

Table 3. Results of the Improved N-queens Kernel Using Shared Memory

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.08	0.27	2.13	15.51	134.23
Speedups	4.63	4.67	4.48	4.38	4.29

Solve the Memory Bank Conflict. In the improved n-queens kernel that uses shared memory, the "ROW" array allocated in the shared memory has two dimensions. The first dimension is a multiple of 32 while the second dimension is not. This memory layout causes bank conflict [18] and hence performance drops when threads access this array. This issue can be addressed by exchanging the first and the second dimension of the array. The performance result is shown in Table 4. The speedups refer to the time ratios of the improved kernel using shared memory with bank conflict (Table 3) versus the new kernel with no bank conflict. On average, eliminating the bank conflict yields a 1.24x speedup.

Table 4. Results of the Improved N-queens Kernel Using Shared Memory with no Bank Conflict

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.06	0.22	1.74	12.82	112.20
Speedups	1.33	1.23	1.22	1.21	1.20

4.2 Reducing Divergence with the IRRC Approach

The loop structures in program code fall into two categories: regular and irregular, as shown in Figure 2. A regular loop has a definite number of iterations while the number of iterations of an irregular loop depends on certain conditions. During each iteration of a loop, threads that take different code branches have to waste time on waiting for others due to the mechanism that GPUs use to handle branch divergence [9].

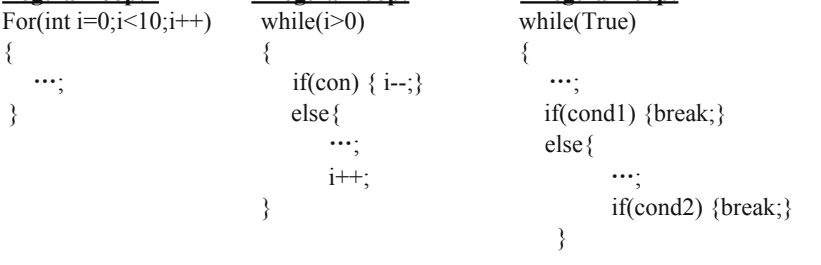


Fig. 2. Regular and Irregular Loop Structure

As shown in the n-queens kernel in Figure 1, the main part of the kernel is an irregular while loop. At each iteration, the threads within a warp begin the computation on their own data which is a sub-problem. When the threads within a warp take different code paths, branch divergence occurs. For the 18-queens problem, we measured the ratio of divergent branches to be 22.6% with the Nvidia Compute Visual Profiler. Moreover, the numbers of iterations of threads increase exponentially with N, so the divergence could lead to a substantial negative impact on performance.

If we can reorganize the work of each iteration to let threads within each warp have more chance to execute together on the same code branches, performance will be better. After studying, we found that the reorganization of work can be done with three methods: adding tail checking to bypass some waiting iterations; combining multiple loop sentences into a single loop sentence; changing an invocation of a kernel in a loop on the CPU side into a loop structure inside the kernel. Since the methods are all related to iteration, we name this divergence-reducing approach the "IRRC" (Iteration-Related Reorganization of Computation) approach. Following the IRRC approach, we devised a new low-divergence n-queens kernel, as shown in Figure 3.

In this low-divergence n-queens kernel, we reordered the condition testing and added a tail check (line 14-16). In the naive n-queens kernel, we followed the pattern of "if cond then X else Y", where part X is much shorter than part Y. At an iteration, if some threads within a warp need to execute part X while others need to execute part Y, the warp has to serially execute part X followed by part Y [9]. Thus, the threads executing part X are idle for a large fraction of time without being able to run possible part Y in the next iteration. In this

```
__global__ void nqueen_kernel_2(*job_data, *results ... )
{
  __const__     tid; //The index of the thread within the block
  __register__  rowIndex, solution;
  __shared__    ROW[MAX_ROW][BLOCK_SIZE];

  each thread fetches a task from job_data into its array ROW[MAX_ROW] [tid];
   for(; rowIndex >= 0; rowIndex--) {
     if ((P = new position) is a valid position ){
       places a queen at P in ROW[rowIndex][tid] and mark the position as occupied;
       if (not reach last row){
           generates ROW[rowIndex+1][tid] based on Row[rowIndex][tid] and the position  P;
           rowIndex++;
           if (row rowIndex has empty candidate positions){
              rowIndex++;
           }
        }
        else{
          solution++;
        }
     }
  }

  reduction of the solutions of the threads within this block;
}
```

Fig. 3. Pseudo Code of the Low-divergence N-queens Kernel

low-divergence implementation, we move the condition testing into the tail of the earlier iteration. Though every thread needs to spend time for parts Y and X, threads are potentially allowed to bypass one waiting iteration, thus the branch divergence is reduced.

With this low-divergence n-queens kernel, the ratio of divergent branches to total branches dwindles to 0.000846% for the 18-queens problem. The performance of this kernel is presented in Table 5. The speedups refer to the time ratios of the improved kernel utilizing shared memory with no bank conflict (Table 4) to this new low-divergence n-queens kernel. On average, a 1.30x speedup is achieved through divergence reduction.

Table 5. Results of the Low-divergence N-queens Kernel

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.05	0.17	1.30	9.60	84.05
Speedups	1.20	1.29	1.34	1.34	1.33

How to Apply the IRRC Approach. Like the approach proposed in [2], the IRRC approach also requires an understanding of the algorithms in the application. The general rule is to reorganize the kernel code inside loops especially irregular loops, such that threads have more chances to take the same code branches. The three methods introduced for reorganization can be used separately or jointly.

4.3 Balancing the Load at Thread Level

As discussed previously, each thread of the n-queens kernel processes a subproblem that has a different number of solutions. Therefore, each threads will execute a different number of iterations. Consequently, the grain sizes of the threads vary. To verify this assumption, we measured the number of iterations and the clock cycles of the 7.41 million GPU threads from the 18-queens problem. The measurement is done inside the for loop of each thread with the low-divergence kernel. The result is as follows.

Table 6. Granularity Variance in the 18-queens Problem

Thread Granularity	Min	Max	Average	Variance	Standard Deviation
Iterations	2.00E+00	4.75E+04	6.51E+03	1.45E+07	3.81E+03
Clock Cycles	1.03E+03	2.80E+07	4.19E+06	6.12E+12	2.47E+06

As shown in Table 6, the n-queens problem exhibits heavy irregularity. GPUs are unable to balance the load at thread level. Because of the converge mechanism for branch divergence [9,10], threads with fewer iterations have to wait for the threads with more iterations within the same warp before exiting a loop structure.

We define the utilization μ_{util} of an SM as follows:

$$\mu_{util} = \frac{\sum_{j=1}^{m} \sum_{i=1}^{n} C(i,j)}{n \times \sum_{j=1}^{m} L(j)} \tag{1}$$

where n denotes the number of threads in each warp, m denotes the number of warps in the SM, $C(i,j)$ denotes the clock cycles used in processing jobs by thread i of warp j, $L(j)$ denotes the clock cycles used by the longest thread in warp j. We sampled 100 warps in the low-divergence n-queens kernel for the 18-queens problem. In these warps, the minimum, maximum and average μ_{util} were 38.39%, 73.92%, and 55.15% respectively. So, the theoretical upper-bound of the speedup with an ideal load-balancing approach is 1.81. Since the granularity of each thread cannot be known prior to the execution, the load balancing of threads can only be done at runtime.

Based on all these analyses, we propose a distributed job pool approach to balance the load at thread level at runtime, as illustrated in Figure 4. Each block in a SM occupies a job pool in global memory that contains a predefined number of jobs. Each thread in a block is assigned an initial job and can get a new job upon finishing. In the end, some threads consume more jobs while others consume fewer, but the overall execution times of all threads within a block/warp are close to each other, hence the load is balanced. The next job in the job pool is controlled using a semaphore. GTX480 supports atomic operations on variables in shared memory or global memory for mutexes.

The pseudocode of the n-queens kernel that utilized this approach is shown in Figure 5. A new variable named *seek* is allocated in the shared memory to point

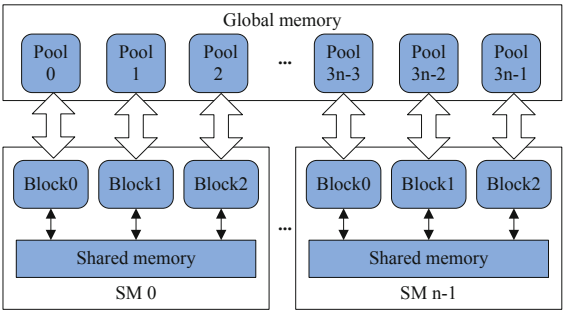


Fig. 4. Distributed job pool Approach

```
1  __global__ void nqueen_kernel_3(*job_data, *results ... )
2  {
3     __const__      tid; //The index of the thread within the block
4     __register__   rowIndex, solution, index;
5     __shared__     ROW[MAX_ROW][BLOCK_SIZE];
6     __const__       upper_bound = the upper bound of the job-pool for this block;
7     __shared__      seek;
8
9     if(tid == 0) { set seek to point to the next new job in the job-pool for this block;}
10    each thread fetches a task from job_data into its array ROW[MAX_ROW] [tid];
11    for(; rowIndex >= 0; rowIndex--) {
12
13       ····· //the same code as in the low-divergence n-queens kernel is omitted
14
15       if (rowIndex == 0) { //current job is done.
16          index = atomicAdd(&seek,1); //get index of new job
17          if (index exceeds pool upper bound)
18             break;
19          else{
20             gets this job by index from the job-pool as the new job of this thread;
21             rowIndex++;
22          }
23       }
24    }
25
26    reduction of the solutions of the threads within this block;
27 }
```

Fig. 5. Pseudo Code of the job pool N-queens Kernel

to the next job in the job pool. The upper part of the code inside the for loop is the same as in the low-divergence n-queens kernel. When a thread finishes its current job (if *rowIndex* == 0), it attempts to fetch a new job in three steps as follows:

- Get the index of the next job through an atomic operation on *seek*.
- Verify this index and break the for loop if the index is out of the range of the job pool (all jobs in the pool are processed).

– Fetch this new job from the job pool, increase *rowIndex* and continue the for loop.

The performance of the n-queens kernel with job pool is presented in Table 7. The speedups refer to the time ratios of the low-divergence kernel (Table 5) to this new n-queens kernel with job pool. All n-queens problem run faster except for the 15-queens problem. The 15-queens problem is a small problem that has a parallelism of only 7432. The job pool approach will decrease the number of threads launched on the GPU since each thread now consumes more than one job. As a result, the number of threads launched for the 15-queens problem cannot fully utilize the hardware capacity, hence the decrease of performance. However, this is not a problem for middle or large scale problems because they have adequate parallelism.

The performance of the 18-queens problem with different sizes of job pool is presented in the following Figure 6. The discussion is as follows.

Overhead Analysis. 1. Time overhead. When one thread does the atomic operation and then fetches a new job, all other threads within its warp will be halted, causing branch divergence. The overhead depends on the clock cycles to fetch a new job including the atomic operation, and the number of times this part of the code is executed. It is mandatory to condense this part of the code to reduce the time overhead. 2. Memory overhead. The job pool can be constructed by dividing the original memory space of data into the number of pools, or by

Table 7. Results of the Distributed job pool N-queens Kernel

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.06	0.16	0.98	6.47	53.61
Speedups	0.83	1.06	1.33	1.48	1.57

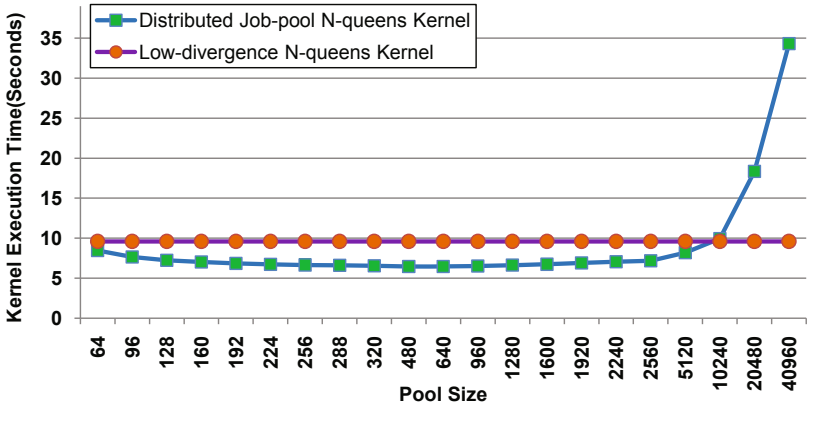


Fig. 6. Execution Time of the 18-queens Problem with Different Sizes of job pool

allocating new memory space. The former has the advantage of saving memory space and memory copying time, therefore is adopted in our solver.

Optimal Size of the Job Pool. The size of the job pool refers to the number of jobs inside. For distributed job pool approach, each block will have a job pool. The size of the job pool is a trade-off: larger pool size may bring a better balance in granularity of threads and thus less waste in waiting. However, the expense to perform load balancing also increases because threads then fetch jobs from the pool for more times, so there is more branch divergence. As shown in Figure 6, the n-queens kernel with distributed job pool runs faster than the low-divergence n-queens kernel for the 18-queens problem with any pool size from 64 to 5120. However, using a pool size like 10240 or larger would cause the overhead of load balancing to outweighs the benefit. In such a case the load balancing approach is no longer profitable.

Distributed Job Pool vs. Monolithic Job Pool. In the distributed job pool approach, there are multiple job pools, one for each block. The total jobs are divided into these job pools at runtime before threads begin taking from them. Therefore, the balance reached is a local balance within each block. We propose another approach where all blocks share a single global job pool. In this monolithic job pool approach, threads from all blocks acquire jobs from that pool, therefore a global balance is reached. However, this approach has more contention for the atomic semaphore than the distributed job pool. Besides, the atomic operation could become expensive because the atomic semaphore needs to become global and resides in the slow global memory. In general, the monolithic job pool approach has its advantages and disadvantages. The result for the n-queens kernel with a monolithic job pool is shown in Table 8. The performance is a little better than using the distributed job pool approach. The Overall Speedups refer to the time ratio of the naive kernel (Table 1) to this new kernel with monolithic job pool.

Table 8. Results of the N-queens Kernel with Monolithic job pool

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.06	0.15	0.86	6.15	51.31
Overall Speedups	6.67	9.73	12.83	12.77	12.94

How to Apply the Job Pool Approach. GPU kernels for solving irregular applications generally exhibit more or less variance in thread granularity. Some show the variance explicitly while others may need some code modification to expose it. The code modification can be made with the three code reorganization methods proposed in section 4.2. In general, the piece of code for load balancing needs to be placed at the end of an irregular loop (refer to section 4.2 for irregular loop), to enable threads to fetch new jobs upon finishing their work. Also, several new variables are needed for the operation of the load balancing code.

4.4 Performance Summary

The following Figure 7 compares all the results. The n-queens kernels with distributed job pool or Monolithic job pool have similar performance, and take the least execution time among all kernels. As shown in Table 8, the average speedup of the Monolithic job pool n-queens kernel over the naive n-queens kernel is around 13 for 17-queens, 18-queens, and 19-queens problem. This shows that appropriate implementation and optimization techniques that consider the characteristics of applications and hardware architectures are crucial for performance of irregular applications. The 15-queens problem and the 16-queens problem have relatively lower speedups of around 7 and 10 respectively, partly because they have smaller parallelism, smaller granularity, and consequently less room to improve.

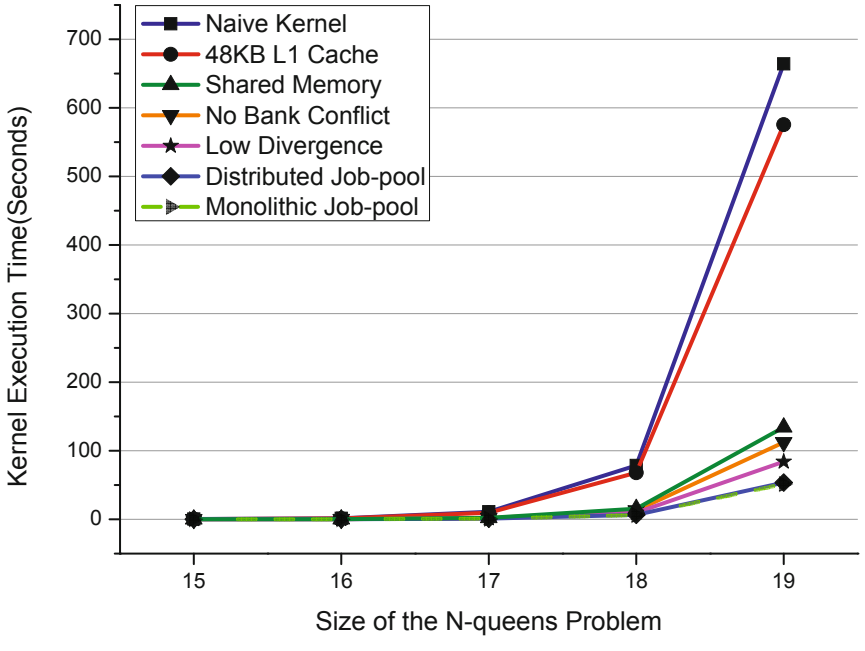


Fig. 7. Execution Time of All Kernels

5 Discussion

Besides the experiments in this work, we also applied the proposed methods on other irregular applications such as an application to calculate the potential distribution of a rectangular waveguide using the montecarlo method. We also got around 1.2x speedup by applying one of the methods to reduce divergence: combining one regular loop with one irregular loop. In general, the proposed approaches in this work can be used separately or combined to reduce divergence.

The proposed distributed or monolithic job pool approaches can balance the load of threads at some time cost and need no extra memory. These approaches are especially useful for coarse-grained threads with remarkable granularity variance. In some dynamic and/or irregular applications, the program size/space can be quite large or unpredictable. Therefore the applications often employ course-grained threads to search in the solution space. The job pool approaches would be useful for such applications.

Although Table 8 shows that the n-queens kernel with a monolithic job pool works faster than that with a distributed job pool, the distributed job pool might be a better choice in some circumstances. In GPUs other than the GTX480 that lack cache, the access to the atomic semaphore in global memory for the monolithic job pool approach might become an expensive operation. Moreover, the distributed job pool approach has better scalability than the monolithic job pool approach. The monolithic job pool approach introduces more contention for the atomic semaphore, and the contention increases with the number of simultaneous threads. Current hardware like Nvidia GPU, AMD GPU and Intel Larrabee can hold more than twenty thousand simultaneous threads on a single chip. We envision that in future that this number will be doubled or tripled often with architectural advancements.

6 Conclusion

We optimized the memory access of the n-queens problem benchmark, and found that the performance was on average 6.34x better with shared memory than using global memory across different sizes of the n-queens problem. Also, we proposed the IRRC (Iteration-Related Reorganization of Computation) approach to reduce branch divergence based on our categorization of the loop structures. By employing this approach, the divergence ratio was reduced remarkably in our experiment. The irregular computation achieved an average 1.30x further speedup over the n-queens kernel that had optimized memory access. Finally, we proposed two approaches to balance the thread load at run time: distributed job pool approach and monolithic job pool approach. These two approaches had similar performance in our experimental evaluation. With the load balancing approaches, the irregular computation ran 1.26x faster on average than the low-divergence n-queens problem kernel. For all the proposed approaches, we introduced their principles and analyzed the factors that affect their effectiveness. Especially, for the two job pool approaches, we made extensive analyses on the time and memory expenses, the optimal size for the job pool, and the performance differences.

Our work is the first to reduce the branch divergence based on our classification of loop structures, and the first to identify and address the load imbalance at thread level at runtime. Together the IRRC approach and the job pool approaches can indeed effectively leverage the impressive computation power of graphics processors. All other irregular applications on GPUs can benefit from the proposed general approaches. As part of our future work, we will apply these

approaches on other dynamic and/or irregular problems, such as sparse matrix-vector multiplication with an unrestricted nonzeros pattern, and the Euler solver sweeping over unstructured meshes. We will keep improving our approaches and devise more general approaches.

Acknowledgment. The authors would like to thank Linghe Kong, Xiaoyang Liu, Sandy Harris and anonymous reviewers for their fruitful feedback and comments that have helped them improve the quality of this work. This research was partially supported by NSF of China under grant No. 61073158.

References

1. Hussein, M., Abd-Almageed, W.: Efficnent Band Approximation of Gram Matrices for Large Scale Kernel Methods on GPUs. In: Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–10. ACM Press, New York (2009)
2. Zhang, E.Z., Jiang, Y., Guo, Z., Shen, X.: Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In: 24th ACM International Conference on Supercomputing (ICS), pp. 115–126. ACM Press, New York (2010)
3. Cederman, D., Tsigas, P.: On Dynamic Load Balancing on Graphics Processors. In: 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, pp. 57–64. ACM Press, New York (2008)
4. Tzeng, S., Patney, A., Owens, J.D.: Task Management for Irregular-ParallelWorkloads on the GPU. In: High Performance Graphics 2010, pp. 29–37. ACM Press, New York (2010)
5. Aila, T., Laine, S.: Understanding the efficiency of ray traversal on GPUs. In: Proceedings of High Performance Graphics 2009, pp. 145–149. ACM Press, New York (2009)
6. Solomon, S., Thulasiraman, P.: Performance Study of Mapping Irregular Computations on GPUs. In: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), pp. 1–8. IEEE Press, New York (2010)
7. Deng, Y., Wang, B.D., Mu, S.: Taming Irregular EDA Applications on GPUs. In: Proceedings of the 2009 International Conference on Computer-Aided Design, pp. 539–546. ACM Press, New York (2009)
8. Vuduc, R., Chandramowlishwaran, A., Choi, J.W., Guney, M.E., Shringarpure, A.: On the Limits of GPU Acceleration. In: Hot Topics in Parallelism (HotPar). USENIX Association, Berkeley (2010)
9. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. J. IEEE Micro. 28, 39–55 (2008)
10. Fung, W.W.L., Sham, I., Yuan, G., Aamodt, T.M.: Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 407–420. IEEE Press, New York (2007)
11. Bell, J., Stevens, B.: A survey of known results and research areas for n-queens. J. Discrete Math. 309, 1–31 (2009)

12. Bozinovski, A., Bozinovski, S.: n-queenss pattern generation: an insight into space complexity of a backtracking algorithm. In: 2004 International Symposium on Information and Communication Technologies, pp. 281–286. Trinity College Dublin, Dublin (2004)
13. Khan, S., Bilal, M., Sharif, M., Sajid, M., Baig, R.: Solution of n-Queen Problem Using ACO. In: IEEE 13th International Multitopic Conference (INMIC), pp. 1–5. IEEE Press, New York (2009)
14. QUEESNTUD project, http://queens.inf.tu-dresden.de/
15. Shu, W., Wu, M.Y.: Asynchronous problems on SIMD parallel computers. J. IEEE Trans. on Parallel and Distributed Systems 6, 704–713 (1995)
16. Blas, A.D., Hughey, R.: Explicit SIMD Programming for Asynchronous Applications. In: IEEE International Conference on Application-Specific Systems, Architectures, and Processors, pp. 258–267. IEEE Press, New York (2000)
17. Cull, P., Pandey, R.: Isomorphism and the n-queenss problem. J. ACM SIGCSE Bulletin 26, 29–36 (1994)
18. NVIDIA CUDA C Programming Guide, http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUDA_C_Programming_Guide.pdf

ParTool: A Feedback-Directed Parallelizer

Varun Mishra and Sanjeev K. Aggarwal

Department of Computer Science and Engineering,
Indian Institute of Technology, Kanpur
varunx@iitk.ac.in, ska@cse.iitk.ac.in

Abstract. We present a tool which gives detailed feedback to application developers on how their programs can be made amenable to parallelization. Also, the tool automatically parallelizes the code for a large number of constructs. Since the tool outputs a parallelized code with OpenMP pragmas, the feedback cycle can be run any number of times till the desired performance is achieved. ParTool also acts as a good learning tool to understand the usage of these parallel constructs and thus enables the developer to write better quality code in the future. Our results show that the use of compiler generated feedback can greatly improve the performance of not only benchmarked code but also real life applications.

Keywords: Automatic Parallelization, compiler feedback, OpenMP, Multicore.

1 Introduction

With the advent of multi-core architectures, the need to fully utilize the capabilities of a computer system has become a topic of great interest among application developers. So far, software has not been able to keep up with the lightning speed with which advancements are being made in the hardware. Since it is difficult to master the skills of manually writing good quality parallel code, other solutions have been sought. Various attempts have been made in the past [1,2,3] to either automate this process of converting sequential programs to parallel or encouraging the use of parallel design patterns, as templates, to develop parallel applications.

Historical auto parallelizing compilers have not found much success in providing speed ups. They seem to do well on benchmarks which are fine tuned and well implemented to enable automatic parallelization but their performance is poor on real life code which is written by application developers. Failure to understand the the overall structure of the program, the algorithm used, or the reason of deploying a particular data structure, prevents these tools from extracting sufficient parallelism from it. Other approaches like use of parallel design patterns also has its limitations: extensibility, the steep learning curve of understanding these patterns and the fact that they cater to only a small subspace of applications. These shortcomings of the current approaches towards parallelization have necessitated the search for other options. We believe that tools which can provide valuable feedback to the user regarding the dependencies which prevent parallelization, can greatly help them to write code which is amenable to automatic parallelization.

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 157–171, 2011.

ParTool, which is built over the ROSE [4,5] compiler infrastructure, inserts OpenMP pragmas in serial code. It performs data dependence analysis provided by ROSE to ascertain whether a loop nest is safe to parallelize. If not, the dependencies which prevented parallelization are emitted in an easy to understand format. This descriptive feedback is very helpful in understanding the dependencies which hinder parallelism and can be used to make suitable modifications to the source code so that these dependencies can be eliminated. ParTool enables the programmer to take a middle path between tedious parallelization and inefficient auto parallelization. We have also built a normal GUI, which makes it easy to inspect the source code, the parallelized code and the emitted feedback. The results show that our approach benefits from the feedback cycle. They support our claim that automatic parallelization is not sufficient for gaining performance improvements and that feedback-directed parallelization can be useful in helping the developers to tap the multi-core resources efficiently.

The rest of this paper is organized as follows. In Section 2, we give a overview of the ROSE compiler infrastructure. Section 3 discusses the details of our feedback-directed parallelizer. In Section 4, we present the results of our work on benchmark and real life applications. Section 5 discusses related work. Lastly, in Section 6 we draw conclusions from our work and discuss future directions.

2 ROSE Compiler Infrastructure

ROSE [4, 5] is an open source compiler infrastructure developed by Lawrence Livermore National Laboratory(LLNL). It acts as a source-to-source translator which takes the source code as input, performs the various operations built into it by the developer and then outputs the transformed code. Its target audience is people who are building tools for analyzing, transforming and optimizing their software. It supports programming languages C/C++ and Fortran. The intermediate representation used in ROSE is an abstract syntax tree(AST) known as *SAGE III*. By using an AST, ROSE preserves the structure of the original code, along with comments and compiler directives enabling it to produce readable output code. No high level constructs are lost and the complexity of the program is not compromised on unparsing.

3 ParTool: Design

ParTool scans over the candidate loop nests and determines whether it could be parallelized by computing its dependence graph. The loops are normalized before constructing its dependence graph. False dependencies are eliminated by suitable code transformations. If no parallelization preventing dependencies remain then the loop is parallelized using OpenMP pragmas. Otherwise, the dependency information is provided to the user. Figure 1 depicts a high level system design of ParTool.

We pre-process the loop nests in the application by normalizing the loop's initialization statement. This is done by promoting the single variable declaration statement outside of the for loop header's initialization statement and rewriting the loop with new index variable. This enables ParTool to work for $C89$ style code.

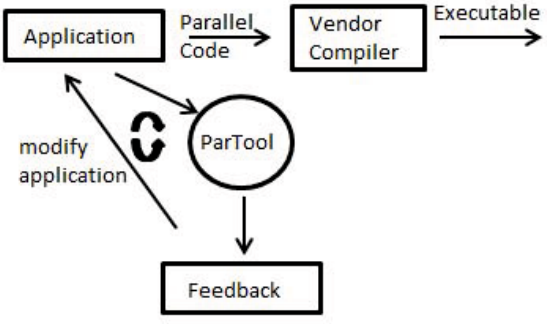


Fig. 1. System Design of ParTool

3.1 Data Flow and Dependence Analysis

Two key components of dataflow analysis, namely def-use(definition usage) analysis and liveness analysis are invoked by ParTool for its usage. They are instantiated for all the functions in the application and are later used while analyzing the application. The def-use analysis of variable references is required for recognizing reduction clauses in OpenMP Pragmas and also as a pre-requisite for invoking liveness analysis, which finds out the variables that can be read before the next re-definition(or write), before each program point. This information is useful while determining whether variables can be deemed private or shared or firstPrivate or lastPrivate, while parallelizing a loop.

The dependence analysis module in ROSE implements the transitive dependence analysis algorithm published by Yi, Adve and Kennedy [7]. The dependence graph is modeled with all statements of the input code as vertices. Directed edges capture dependencies among the statements. The dependence analysis in ParTool classifies the data dependencies as one of the following: true data dependence, anti dependence, output dependence, i/o dependence or scalar dependence. This gives the user a clear idea of what is the nature of the dependency and what are the variables/function calls that are causing it.

3.2 Attaching OpenMP Pragma

After the dependence analysis flags a loop nest to be safe for parallelization, the tool proceeds to attach the suitable pragma. This requires classifying all the variables accessed inside the loop as either shared/private/lastPrivate/firstPrivate or a reduction.This classification makes use of the liveness analysis to determine variables that are live coming into the loop and live going out of the loop. For recognizing reduction statements, we require def-use chains: a variable is a reduction candidate if it is accessed just once in the loop, e.g. $x++/x* = 2$ or exactly twice. e.g. $x = x * 2, x = x + y$.

After variable classification, we generate the string which represents the OpenMP pragma, which is of the form:

#pragma omp parallel for <list of classified variables>

We also support the specification of any of the OpenMP scheduling methods: static, guided, dynamic, auto and runtime. User can choose a flag on commandline to either pick a default schedule for all the parallelized loops in his application or to pick a custom schedule for every parallelized loop separately. The string for scheduling is automatically generated and appended to the OpenMP pragma. So the most general OpenMP pragma attached by ParTool looks like:

#pragma omp parallel for <list of classified variables> schedule(type)

where classified variables are divided into private, lastPrivate, firstPrivate and reduction clauses. The schedule type is one among those defined by OpenMP.

3.3 Feedback

The feedback mechanism of ParTool allows user to understand the loop dependencies which prevent parallelization and assists in removing them. ParTool prints the feedback for every *for loop* in the input code. If a loop is free of any dependencies, it is parallelized and information regarding this is emitted in feedback. Loops which do not posses a canonical form:

for(init expr; test expr; increment expr)

are skipped and reported in the feedback. For loops that could not be parallelized, all the remaining dependencies are printed in a neat manner, sorted according to line number of the source node. Each such dependency from statement s_1 on line l_1 to statement s_2 on line l_2, caused by variable v_1 in s_1 and variable v_2 in s_2 is printed in this format:

depType* l_1* $v_1 \longrightarrow l_2$* v_2

A summary of parallelization efforts is printed at the end of the feedback. It includes total number of loops encountered, number of non-canonical loops that were skipped, number of loops that were parallelized, count and line numbers of the loops that remained to be parallelized. These statistics give a nice overview of the performance of ParTool on the input application and acts as a measure of the extent of work needed to gain desired performance improvement.

3.4 Usefulness of the Feedback Mechanism

In this section, we provide snippets of code and feedback, which underline the fact that feedback is important in extraction of parallelism.

Example 1. We first present a loop nest, from Ising model of ferromagnetism, which was critical in gaining performance improvement. Program 1.3 shows the original loop nest and program 1.1 shows the ParTool feedback.

The feedback shows that the first few lines (which calculate indices using a call to $rn()$: a random number generator), do not produce any data dependencies. They repeatedly use the scalar variables $temp1$ to $temp4$ for generating indices into the $spins$ matrix. This hints towards splitting the loop into two and finding all the required array indices apriori. It turns out that after this simple modification, the first loop can be easily parallelized and as the Figure 4b suggests, we obtained considerable performance improvement.

Program 1.2 shows the random number generator used in this program and program 1.4 depicts the modified code which made use of the feedback to make the loop nest amenable to parallelization.

```
 Could not parallelize a loop at line:76 due to the following dependencies:

* Anti Dependence *Line 84:((spins[temp1])[iy]) --> Line 89:(spins[ix])[iy]
* Anti Dependence *Line 84:((spins[temp1])[iy]) --> Line 97:(spins[ix])[iy]
* Anti Dependence *Line 84:((spins[ix])[temp2]) --> Line 89:(spins[ix])[iy]
* Anti Dependence *Line 84:((spins[ix])[temp2]) --> Line 97:(spins[ix])[iy]
* Anti Dependence *Line 84:((spins[temp3])[iy]) --> Line 89:(spins[ix])[iy]
* Anti Dependence *Line 84:((spins[temp3])[iy]) --> Line 97:(spins[ix])[iy]
* Anti Dependence *Line 84:((spins[ix])[temp4]) --> Line 89:(spins[ix])[iy]
* Anti Dependence *Line 84:((spins[ix])[temp4]) --> Line 97:(spins[ix])[iy]
* Anti Dependence *Line 85:((spins[ix])[iy]) --> Line 89:(spins[ix])[iy]
* Anti Dependence *Line 85:((spins[ix])[iy]) --> Line 97:(spins[ix])[iy]
* Output Dependence *Line 89:(spins[ix])[iy] --> Line 89:(spins[ix])[iy]
* Output Dependence *Line 89:(spins[ix])[iy] --> Line 97:(spins[ix])[iy]
* Scalar Dependence *Line 91:pos --> Line 91:pos
* Scalar Dependence *Line 91:pos --> Line 99:pos
* Back-Scalar Dependence *Line 99:pos --> Line 91:pos
* Output Dependence *Line 97:(spins[ix])[iy] --> Line 97:(spins[ix])[iy]
* Scalar Dependence *Line 99:pos --> Line 99:pos
...
...
... and more dependencies
```

Program 1.1. Corresponding Feedback (Ising)

```
/* RANDOM NUMBER GENERATOR */
double rn()
{
  class boost::uniform_real< double  > uni_dist((0),(1));
  class boost::variate_generator< boost::mt19937 & , boost::uniform_real<
       double  >  > uni(gen,uni_dist);
  return uni();
}
```

Program 1.2. Code for random function (Ising)

```
/* ORIGINAL LOOP */
for(i=0;i<n*n;i++)
{
    ix = n*rn();
    iy = n*rn();
    temp1 = (ix+1)%n;
    temp2 = (iy+1)%n;
    temp3 = (ix-1+n)%n;
    temp4 = (iy-1+n)%n;
    /*line 84*/fac = (spins[temp1][iy] + spins[ix][temp2] +
        spins[temp3][iy] + spins[ix][temp4]);
    fac = spins[ix][iy]*fac;
    ind = fac/2 + 2;
    if(ind<=2)
    {
          spins[ix][iy] = -1*spins[ix][iy];
          energy = energy+2*J*fac;
          pos=pos+1;
          mag=mag+2*(spins[ix][iy]);
```

```
    }
    else {
          if((rn())<ex[ind])
          {
                spins[ix][iy]=-1*spins[ix][iy];
                energy+=2*J*fac;
                pos=pos+1;
                mag=mag+2*(spins[ix][iy]);
          }
    }
}
```

Program 1.3. Original Loop Nest (Ising)

```
int ix[n], iy[n];
int temp1[n],temp2[n],temp3[n],temp4[n];
int rns[n];

/* MODIFIED LOOP PARALLELIZED BY PARTOOL */
#pragma omp parallel for
for(i=0;i<n*n;i++)
{
    ix[i] = n*rn();
    iy[i] = n*rn();
    temp1[i] = (ix[i]+1)%n;
    temp2[i] = (iy[i]+1)%n;
    temp3[i] = (ix[i]-1+n)%n;
    temp4[i] = (iy[i]-1+n)%n;
    rns[i] = rn();
}
 for(i=0;i<n*n;i++)
{
    fac = (spins[temp1[i]][iy[i]] + spins[ix[i]][temp2[i]] +
        spins[temp3[i]][iy[i]] + spins[ix[i]][temp4[i]]);
    fac = spins[ix[i]][iy[i]]*fac;
    ind = fac/2 + 2;
    if(ind<=2)
    {
          spins[ix[i]][iy[i]] = -1*spins[ix[i]][iy[i]];
          energy = energy+2*J*fac;
          pos=pos+1;
          mag=mag+2*(spins[ix[i]][iy[i]]);
    }
    else if(rns[i]<ex[ind])
    {
                spins[ix[i]][iy[i]]=-1*spins[ix[i]][iy[i]];
                energy+=2*J*fac;
                pos=pos+1;
                mag=mag+2*(spins[ix[i]][iy[i]]);
    }
}
```

Program 1.4. Modified Loop parallelized using ParTool (Ising)

Usage of linear computation in the array subscript was the real cause of failed parallelization. Had it not been for the feedback mechanism, the user would have to manually go through the entire loop nest and look for additional causes of data dependence where none existed.

```
Could not parallelize a loop at line:102 due to the following dependencies:

* Output Dependence * Line 124:output[(i * 2400) + j] --> Line 124:output[(i *
    2400) + j]
-------------------------------------------------------
 Automatically parallelized a loop at line 103
-------------------------------------------------------
```

Program 1.5. Corresponding Feedback (Mandelbrot)

```
for(i=0; i<2400; ++i) {/*line 102*/
   for(j=0; j<2400; ++j) {/*line 103*/
      c_real = real_min + i * real_range;
      c_imag = imag_min + j * imag_range;
      z_real = 0.0;
      z_imag = 0.0;
      for(counter = 0; counter < MAX_ITER; ++counter) {
            z_current_real = z_real;
            z_real = (z_real * z_real) - (z_imag * z_imag)+
                c_real;
            z_imag = (2.0 * z_current_real * z_imag)+c_imag;
            z_magnitude = (z_real * z_real) + (z_imag * z_imag
                );

            if(z_magnitude > RADIUS_SQ) {
                  break;
            }
      }  //end for
      output[i*2400+j] = (int)floor(((double)(255 * counter))
          / (double)MAX_ITER);
   } // end for
} // end for
```

Program 1.6. Original Loop Nest (Mandelbrot)

Example 2. We present code snippets from the MANDELBROT benchmark code, which calculates a fractal structure in the complex plane. Program 1.6 shows a loop nest which calculates pixel values and stores them in the *output* array. When this program is given as input to ParTool, we get the feedback as shown in program 1.5. The inner loop with index j was parallelized, while the outer loop had a single dependency. The feedback states that *output dependence* while writing to the *output* array, is preventing parallelization. A quick scan of the loop nest is enough to realize that there is no output dependence since the array subscript $i * 2400 + j$ is unique for different values of i and j (because both i and j are bounded above by 2400).

```
#pragma omp parallel for private (c_real,c_imag,z_real,z_imag,
    z_magnitude,z_current_real)
for(i=0; i<2400; ++i) {/*line 102*/
   #pragma omp parallel for private (c_real,c_imag,z_real,
       z_imag,z_magnitude,z_current_real) firstprivate(i)
   for(j=0; j<2400; ++j) {/*line 103*/
      c_real = real_min + i * real_range;
      c_imag = imag_min + j * imag_range;
      z_real = 0.0;
      z_imag = 0.0;
      for(counter = 0; counter < MAX_ITER; ++counter) {
            z_current_real = z_real;
            z_real = (z_real * z_real) - (z_imag * z_imag)+
                c_real;
            z_imag = (2.0 * z_current_real * z_imag)+c_imag;
            z_magnitude = (z_real * z_real) + (z_imag * z_imag
                );

            if(z_magnitude > RADIUS_SQ) {
                  break;
            }
      }  //end for
      output[i*2400+j] = (int)floor(((double)(255 * counter))
          / (double)MAX_ITER);
   } // end for
} // end for
```

Program 1.7. Modified Loop parallelized using ParTool (Mandelbrot)

Since ParTool is a source-to-source translator, we can safely comment out the problematic line and obtain program 1.7, which parallelizes the outer loop as well. The line causing the false *output dependence* can now be re-inserted into the code. If our tool produced an executable, then we could not have used this approach. The only option would have been to figure out the OpenMP pragma manually and insert it into the source code itself. This example shows us the utility of two key features of ParTool: feedback mechanism and source-to-source translation.

3.5 CommandLine Options

We have built several useful options into ParTool which assist the user to tailor the quality of parallelized code and better understand the feedback generated by ParTool. These flags can be set on the commandline either alone or in combination to get the desired results. Table 1 shows the list of flags which can be used from the command line interface.

3.6 Code Characteristics Suitable for ParTool

The functionality implemented in ParTool works well for loops with countable iteration space. Loops can have branch conditions, local variables and complex nested structure.

Table 1. List of commandline options

flag	description
-par:debug	runs ParTool in a debugging mode. Prints additional info that help understand the loop nests in greater depth
-par:printStmt	prints statements too and not just line number while printing dependencies
-par:stmtWise	groups the dependences statement-wise according to source node
-par:Schedule	option to provide the schedule type for every loop one by one
-par:defSchedule	option to accept a default schedule type for every loop
-par:focus	allows to focus on the feedback of a single loop at a time
-par:stats	prints a statistical summary of parallelization efforts
-annot filename	specify an annotation file for semantic abstraction support
-edg:w	suppress the warnings produced by the EDG parser
–help	prints the default help information for a typical ROSE translator

Function calls, with known semantics, do not prevent the parallelization process. ParTool directly uses the dependence graph computed by ROSE dependence analysis [7] and then determines the dependencies which are applicable to the target loop. ParTool works for arrays declared as pointers, if pointer syntax is not used inside loops. For example, an integer pointer allocated memory using $malloc()$, is suitable, if the elements are referred as $a[i]$ and not $*(a + i)$ inside a loop.

ParTool attempts to parallelize all the loops in the program without doing a Profitability Analysis. This may lead to a degradation of performance due to synchronization overheads. We have implemented *coarse-grain parallelism* by parallelizing only the outermost loop in a loop nest. OpenMP pragmas are attached to the inner loops as well but they are automatically commented by ParTool. The user can then invoke a profiling tool to determine whether it is profitable to parallelize the entire loop nest or not.

The array subscripts with indirect references (such as $a[b[i]]$) are not handled by ParTool. High-level library constructs like STL vectors/lists and user-defined data structures hinder parallelism, since their semantics are not well understood. ParTool does not implement speculative parallelization at runtime, which is suitable for irregular applications [8].

4 Results

We have tested ParTool on both benchmarked codes and real life applications built by engineering students, with no or little background in parallel programming. We ran all the tests on a 6 processor, 24 core 2.66 GHz Intel system running RHEL Server release 5.3. It had 8GB internal memory. We compared the speedups obtained by ParTool from the final modified application after incorporating the feedback with that of ParTool with the original applications. We also obtained the speedups from Intel icc 9.0 compiler. The

command used to parallelize using icc was ./icc -parallel -par-report1 <input file>. The results clearly underline the significance of feedback in performance gain as we witness a drastic improvement after using feedback generated from ParTool.

The results are divided in 3 sections: NAS parallel benchmarks [9] are evaluated in Section 4.2, a couple of real life examples are stated in Section 4.3 and a summary of other benchmark codes is presented in Section 4.4.

4.1 Comparison of Parallelization Performance

In some cases, as will be evident in the results section, the default performance (without feedback) of ParTool is better than that of Intel compiler. Also there are cases, for which Intel compiler performs better.

Better performance of Intel compiler can be because of performing both parallelization and data locality optimization at the same time. ParTool does not perform such optimizations since the ROSE framework for them is currently under development and breaks down too often. In some cases, ParTool performs better since it is able to extract parallelism from loops with function calls. The user can provide annotation files mentioning the variables that are read or written by the function. Intel compiler, on the other hand, may make safe assumptions if it is unable to determine the side-effects from inter-procedural analysis. We also witnessed cases, where the Intel compiler could not parallelize a loop, citing insufficient computational work. ParTool was successful in extracting parallelism and performance gains in such cases.

4.2 NAS Parallel Benchmarks

The NAS Parallel Benchmarks v2.3 are a set of benchmarks from Computation Fluid Dynamics(CFD) domain. They are developed and maintained by the NASA Advanced Supercomputing (NAS) Division based at the NASA Ames Research Center. They are used to solve systems of partial differential equations that model the dynamics of a physical system.

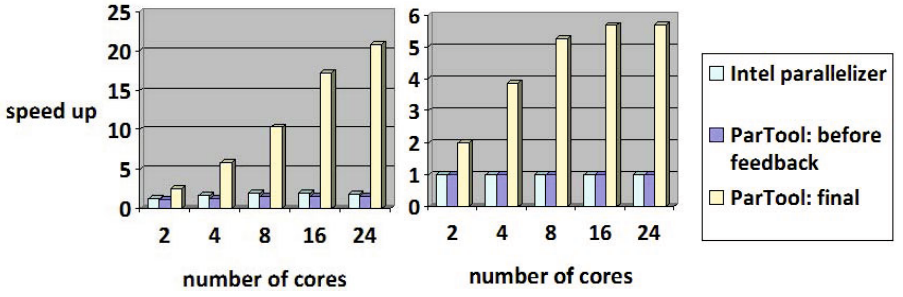
LU. This benchmark solves a synthetic system of nonlinear PDEs using an algorithm involving symmetric successive over-relaxation (SSOR) solver kernels. Figure 2a depicts the 'speedups versus number of cores' chart, with relative performances of ParTool and Intel icc compiler. Without feedback, both the tools achieved a speedup of nearly 2, but with feedback, ParTool performed significantly better.

BT. This benchmark solves a synthetic system of nonlinear PDEs using an algorithm involving block tridiagonal kernels. The relative speedups are presented in Figure 2b, which suggest that both ParTool and icc were unable to extract any parallelism from the benchmark code but with feedback, ParTool was able to achieve drastic improvement.

MG. This code approximates the solution to a three-dimensional discrete Poisson equation using the V-cycle multigrid method. Figure 3 clearly suggests that feedback from ParTool was essential in improving the execution time of the application.

4.3 Real Life Examples

We provide results of parallelization efforts on two real life applications taken from engineering students at Indian Institute of Technology, Kanpur.



(a) NAS LU Solver (b) NAS BT Solver

Fig. 2. Speed-up vs Number of Cores

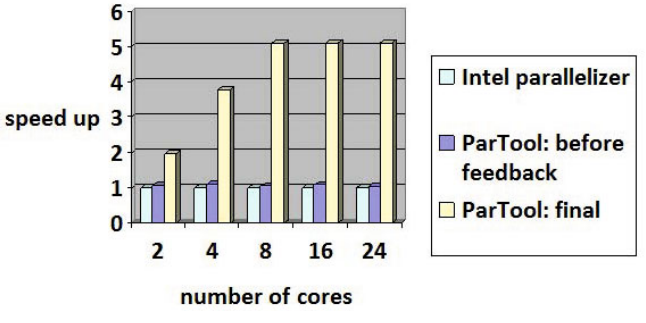
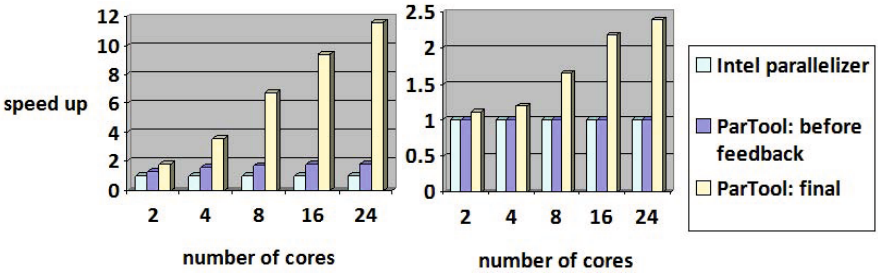


Fig. 3. NAS MG solver

Smoothed Particle Dynamics. This code presents a mesh free particle method (MPM) called Smoothed Particle Hydrodynamics and is used to simulate fluid dynamics problems. The entire mass of fluid is represented in form of particles. Each particle represents a cluster of actual liquid molecules (represented by variables in the code). Standard fluid mechanics equations are numerically solved and the solution gives the value of field variables (velocity, density, acceleration etc.) at each time step. It is an advancement over the traditional Computational Fluid Dynamics method and can be applied in a very vast number of problems.

Results in figure 4a show that the code was not amenable to parallelization until the ParTool feedback was used to modify the loop nests.

2d Ising Model of Ferromagnetism. This application implements the two-dimensional ising model, which is a mathematical model of ferromagnetism in statistical mechanics. Discrete variables named *spins* are arranged in form of a lattice and the aim is to find their phase changes in the simplified Ising model, to understand phase changes in real substances. We were successful in extracting parallelism by using ParTool, as depicted in this figure 4b.



(a) Smoothed Particle Dynamics (b) 2d Ising model of Ferromagnetism

Fig. 4. Results for Real Life Examples

4.4 Other Kernels

We tested ParTool on a variety of popular code which have been used by auto parallelizers. Our results demonstrate that ParTool was able to successfully parallelize them and match the performance of icc. In most cases, feedback was not required to extract parallelism to the maximum extent possible. Description of these codes is as following:

1. **ADI:** an iterative method to solve a system of linear equations
2. **GEMVER:** a linear algebra kernel used in matrix bi-diagonalization and tri-diagonalization. It solves the equation- $\boldsymbol{B = A + u_1 * {v_1}^T + u_2 * {v_2}^T}$
3. **MANDELBROT:** Solves for the mandelbrot set, which is a fractal structure defined in the complex plane by the following equation: $z_n = {z_{n-1}}^2 + z_0$. The set itself is the area where $\lim_{n\to\infty} z_n < \infty$
4. **JACOBI-2D:** iterative solver for a matrix which averages the point's value with its 4 neighbors, until the maximum change reaches a threshold
5. **JACOBI-1D:** iterative solver for an array which averages the point's value with its 2 neighbors, until the maximum change reaches a threshold
6. **CORCOL&COVCOL:** part of Principal Component Analysis(PCA) benchmark. PCA [10] is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of uncorrelated variables called principal components.

Figure 5 provides the relative speed-ups obtained for these benchmark codes.

5 Related Work

Various tools have been built in the past which aim to extract parallelism from sequential code. PIPS [11] is a tool for source to source program optimization, program compilation and auto parallelization. PLUTO [12] is another tool which works on C programs to parallelize loops with affine array access patterns. Cetus [13] is a compiler infrastructure built at Purdue University which supports source-to-source parallelization. However, these tools do not have feedback mechanism and perform at the same level as icc.

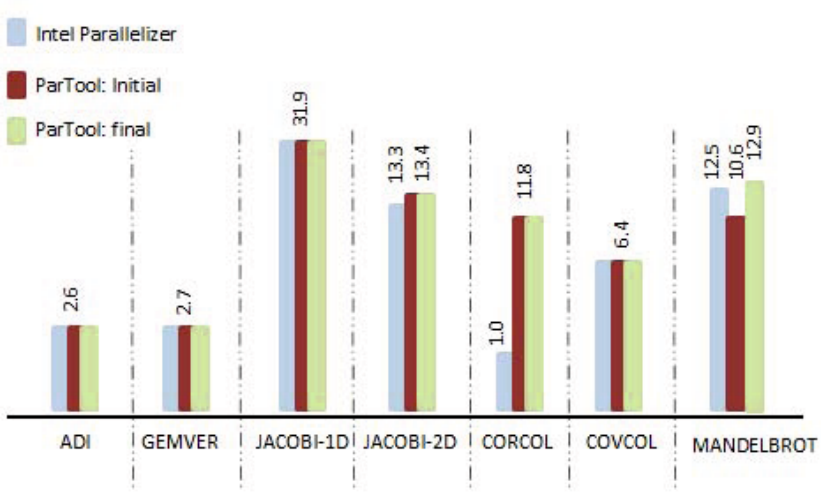


Fig. 5. Summary of Results for other Benchmark Codes (not drawn to scale)

5.1 Comparison of GAP with ParTool

Guided Auto-Parallelization(GAP) [14] is a tool by Intel which guides the user to make changes in their code by offering certain diagnostic advice. It is closest to the work done by us. Here we compare the features of GAP with those of ParTool below.

1. **type of feedback:** GAP provides only diagnostic advice related to usage of local variables and inclusion of compiler flags. In addition, it may suggest a compiler pragma whose correctness has to be manually verified. Whereas, the feedback provided by ParTool comprises of loop dependencies which prevented parallelization. ParTool works directly at the level of target loops which are to be parallelized but GAP gives selective suggestions which may not prove sufficient to understand/remove the dependencies hindering parallelism. For example, for program 1.8, when GAP is unable to parallelize a loop, it merely asks the user (in the output 1.9) to manually verify if there are data dependencies or not. Unlike ParTool, it does not provide the remaining dependence graph to the user, which can be used to suitably modify the code.
2. **parallelization:** GAP does not parallelize the program. It only provides pointers to make the code amenable to parallelization. User has to compile the program using Intel compiler with $-parallel$ flag to actually parallelize the program. So, in effect, it is a two-step process. In contrast, the feedback and parallelization process in ParTool take place simultaneously.
3. **Type of output:** ParTool provides a source code as output, annotated with OpenMP pragmas. The user can further modify the code to extract more parallelism. Whereas, Intel & GAP directly give an executable, so the option of further modifying the parallelized code cannot be exercised.

```
/* Calculate the m * m correlation matrix. */
for (j1 = 1; j1 <= m-1; j1++){
        symmat[j1][j1] = 1.0;
        for (j2 = j1+1; j2 <= m; j2++)  {
                symmat[j1][j2] = 0.0;
                for (i = 1; i <= n; i++)
                      symmat[j1][j2] += ( data[i][j1] * data[i
                         ][j2]);
              symmat[j2][j1] = symmat[j1][j2];
        }
}
```

Program 1.8. Sample Input Code for GAP

```
Insert a "#pragma parallel" statement right before the loop at line 151 to
    parallelize the loop. [VERIFY] Make sure that these arrays in the loop do
    not have cross-iteration dependencies: symmat. A cross-iteration dependency
     exists if a memory location is modified in an iteration of the loop and
    accessed (by a read or a write) in another iteration of the loop.
```

Program 1.9. Sample Feedback of GAP

These differences in the quality of feedback and output suggest that ParTool is more suited to be used as a *feedback-directed parallelizer* than GAP. The latter is only useful for an experienced programmer, for pointing out the loops that require manual parallelization efforts.

6 Conclusion

ParTool is a useful tool for extracting parallelism from C/C++ code. The results support our claim that feedback-directed parallelization is a promising approach to achieve drastic improvements on not only benchmarked code but also real life applications. ParTool helps avoid the steep learning curve involved in understanding parallel programming paradigm and dramatically reduces the effort required to parallelize applications manually. The tool generated feedback helps the user to better understand the dependence relations among instructions in code segments.

ParTool implements the basic functionalities of a feedback-directed parallelizer and is a work in progress. Further improvements to the tool can be made. Some of them are listed here.

- Support other popular programming languages like Java and Fortran
- Include profiling information to identify hotspots in the application so that programmer can channelize his efforts of extracting parallelism
- Provide an estimated speed-up after parallelizing a particular loop nest using tool feedback
- Support additional OpenMP attributes such as synchronization clauses and data copying clauses

References

1. Kennedy, K., et al.: Interactive Parallel Programming using the ParaScope Editor. IEEE Transactions on Parallel and Distributed Systems, 329–341 (July 1991)
2. Smith, K., et al.: Incremental dependence analysis for interactive parallelization. In: Proceedings of the 4th International Conference on Supercomputing, Amsterdam, pp. 330–341 (June 1990)
3. Tournavitis, G., Franke, B.: Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In: PACT 2010 (2010)
4. Quinlan, D.: ROSE: Compiler Support for Object-Oriented Frameworks. In: Proceedings of Conference on Parallel Compilers (CPC2000), Parallel Processing Letters, vol. 10. Springer, Heidelberg (2000)
5. Quinlan, D., Schordan, M., Yi, Q., de Supinsk, B.: Semantic-Driven Parallelization of Loops Operating on User-Defined Containers. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 524–538. Springer, Heidelberg (2004)
6. ROSE User Manual, `http://rosecompiler.org/ROSE_UserManual/ROSE-UserManual.pdf`
7. Yi, Q., Adve, Kennedy.: Transforming loops to recursion for multi-level memory hierarchies. In: ACM-SIGPLAN Conference on Programming Language Design and Implementation, Vancouver, Canada (June 2000)
8. Pingali, K., et al.: Scheduling Strategies for Optimistic Parallel Execution of Irregular Programs. In: Symposium on Parallelism in Algorithms and Architectures, Munich, Germany (June 2008)
9. Bailey, D., Barszcz, E., et al.: The NAS Parallel Benchmarks. Int'l Journal of Supercomputer Applications 5(3), 66–73 (1991)
10. Principal Component Analysis, `http://en.wikipedia.org/wiki/Principal_component_analysis`
11. Ancourt, C., et al.: PIPS: a Workbench for Program Parallelization and Optimization. In: European Parallel Tool Meeting 1996 (EPTM 1996), Onera, France (October 1996)
12. PLUTO - An automatic parallelizer and locality optimizer for multicores, `http://pluto-compiler.sourceforge.net/`
13. Dave, C., et al.: Cetus: A Source-to-Source Compiler Infrastructure for Multicores. IEEE Computer 42(12), 36–42 (2009)
14. Guided Auto-Parallel (GAP), `http://software.intel.com/en-us/articles/guided-auto-parallel-gap/`

MT-Profiler: A Parallel Dynamic Analysis Framework Based on Two-Stage Sampling

Zhibin Yu, Weifu Zhang, and Xuping Tu

Service Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, China, 430074
{yuzhibin,xptu}@mail.hust.edu.cn

Abstract. Dynamic instrumentation systems offer a valuable solution for program profiling and analysis, architectural simulation, and bug detection. However, the performance of target programs suffers great losses when they are instrumented by these systems. This issue is mainly caused by the resource contention between the target programs and the instrumentation systems. As multi-core processors are becoming more and more prevalent in modern computing environments, more hardware resource parallelism is provided for software to exploit. In this paper, we propose to leverage the abundant computing resources of multi-core systems to accelerate program instrumentation. We design and implement a Multi-Threaded Profiling framework named MT-Profiler to dynamically characterize parallel programs. The framework creates instrumenting and analysis code slices which can run along with application threads in parallel on different cores. To further reduce the overhead of dynamic instrumentation, MT-profiler employs two-stage sampling scheme with several optimizations. It is implemented on DynamoRIO which is an open-source dynamic instrumentation and runtime code manipulation system. The performance of our MT-Profiler is evaluated by using the NPB3.3 OPENMP test suite. The results demonstrate that the MT-Profiler obtains 3 to 17 times speedup compared with the DynamoRIO perfect profiler while the average accuracy is over 80%.

Keywords: Multi-core, Multi-thread, Profiling, Sampling.

1 Introduction

Currently, dynamic instrumentation is a popular approach used for program profiling and analysis, architecture simulation, and bug detection. Dynamic instrumentation systems, such as Pin[13], DynamoRIO[3], and Valgrind[15], adopt dynamic compilation to instrument executables when they are running. The common implementation mechanisms of these systems are performed by copying one basic block of the application code at a time into code cache where the block is executed, patching stubs to regain control, and trace linking. The systems also provide abundant and powerful APIs for tool developers to examine and manipulate

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 172–185, 2011.

applications at runtime to glean their interested information. There are many successful dynamic analysis tools have been built based on these dynamic instrumentation systems. However, target programs instrumented by these dynamic analysis tools often suffer several orders of magnitude performance degradation. Precisely, dynamic instrumentation based systems often have overheads varying from 10% for very light instrumentation to 1000x slowdown if the user inserts very sophisticated and time-consuming functionality[3] [10] [13] [18] [20] .Therefore, the overhead of dynamic instrumentations severely limits the usage of them.

The primary cause of the slow speed of instrumented systems is that the target programs and the instrumenting codes contend the same set of computing resources. Time sharing is a common mechanism used in the systems. The more information needs to be profiled, the more time spends on the instrumentation and analysis code, leading to much lower overall performance of target programs. Therefore, it is a good idea to run the instrumentation and analysis code on different computing resources from the ones used by target programs for acceleration if possible.

Recently, microprocessor designers have packed more and more homogeneous or heterogeneous cores on a single chip. These multi-core architectures provide more hardware resources and parallelism for software to exploit than before, which presents us an opportunity to accelerate instrumentation based programs. On the other hand, in multi-core systems, software designers must adopt multi-threaded programming models to take full advantage of the hardware parallelism. However, multi-threaded programming is often inherently complicated, such as deadlock and competition etc. As a result, there is an urgent need for dynamic analysis tools with reasonable overheads to support profiling and tuning of multithreaded programs. Netherless, profiling and tuning of multithreaded programs based on dynamic instrumentation still suffer significant performance losses on multi-core platforms.

In this paper, we propose to harness one or more separated cores to run the instrumenting and analysis code to speedup the instrumentation based systems. We design and implement a framework named MT-Profiler to dynamically analyze the characteristics of parallel programs. Although many scholars have turned to sampling-based instrumentation[1] [11] [7] [8] or exploit thread level parallelism to solve the performance problem of dynamic analysis systems[10] [17] [19], there are significant differences between our methods and those of others. In our framework, we adopt two-stage sampling scheme. At first stage, we use hardware performance monitoring units to sample the execution of applications and instrument checking code. It is similar to[1], but we dont́ need recompile applications and the overhead is low. At second stage, we use the checking code as a software sampler. It checks the sample condition and decides whether to fork a child. If the checking code forks a child, the child will be instrumented analysis code and executed in parallel with the application's execution. In particular, our framework has the following advantages:

– **The overhead is low.** We decouple the instrumenting and analysis code from the application code, so they can be executed in parallel on multi-core processors.
– **The framework is flexible.** With the two-stage sampling strategy, it is flexible and tunable, allowing coarse and fine-grained analysis with acceptable overhead and accuracy.

The rest of this paper is organized as follows. Section 2 discusses the related work. The design and implementation details are described in Section 3. Section 4 presents the performance evaluation and demonstrates the effectiveness of our system. Finally, in Section 5, We conclude this work and introduce the future work.

2 Related Work

Profiling systems based on dynamic instrumentation[6] often suffer high overhead. There are many studies focusing on reducing the overhead of profiling. In this section, we briefly review those work related to our framework.

One popular way to reduce the overhead of profiling is sampling. Arnold et al. suggest approximating the calling context tree by sampling global or individual function (or method) counts[2] [12]. The Arnold-Ryder framework samples information by using global counters in checking code at all procedure entries and loop back-edges [1] [4]. It maintains two versions of instrumented code, one for checking and the other performs the real profiling work. When a counter in the checking code decreases to zero, it executes the instrumented analysis code to collect profiling information, and then resets the counter and transfers back to checking version. Hirzel et al. enhance the Arnold-Ryder framework by reducing the number of checks to reduce the overhead of sampling[10].

Another approach to reduce the overhead of profiling is to exploit thread parallelism by utilizing the multi-core systems. Shadow Profiling harnesses hardware parallelism on multi-core systems by creating shadow processes which are instrumentation processes and executed parallel with the original application[16] [14]. By periodic creating shadow process, it decouples the execution of instrumentation code from application code. Our framework is similar to Shadow Profiling, but we adopt two stages sampling which make our framework is more flexible and portable compared with other dynamic instrumentation systems. SuperPin uses a similar approach to Shadow Profiling, but it deterministically replicates the full program execution[19]. PiPA[21] decouples the process of profiling into two stages which are collecting raw profiling information and analysis. It performs a low overhead profiling to collect raw information, and uses a software pipeline which has several stages and each stage has multiple threads to reconstruct the full profiles for analysis. It parallelizes the analysis process and presents a compact profiling format to reduce the overhead of collecting raw data, but it will be more efficient by using our framework to parallelize the profiling at the data collecting stage. Jungwoo Ha et al. [9] present a similar approach to PiPA. It presents Cache-friendly Asymmetric Buffering (CAB) to improve the efficiency

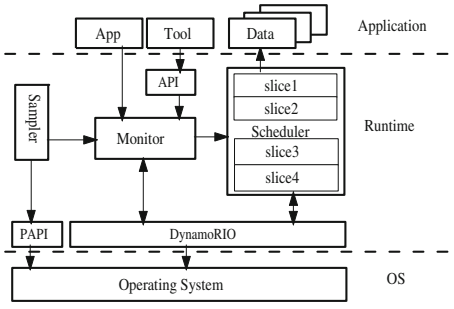


Fig. 1. Architecture overview of MT-Profiler

of buffer for communicating event data from application threads to analyzer threads. It also supports both exhaustive and sampling analysis.

Finally, hardware support is also a way to reduce the overhead of profiling. Many researchers reduce the overhead of profiling by sampling the information monitored by hardware performance monitoring units (PMU)[11] [7]. In our framework, we also use PMU as the first stage sampling strategy. ProfileMe leverages hardware support for profiling the path of specific instructions through the pipeline[8]. With the hardware buffering, instrumentation support, and augmenting each cached word with one extra state bit, HeapMon implements a low-overhead memory-related bug detection system[17]. iWatcher leverages hardware assisted thread-level speculation (TLS) to reduce the overhead of monitoring program locations[22]. Our framework is similar to software TLS, it also exploits threaded parallelism by dynamically creating threads to execute instrumentation codes without hardware support.

3 The Design of MT-Profiler

In this section, we firstly describe the overview of our framework. Then we present the design and implementation detail of each component.

3.1 System Overview

Our goal is to significantly reduce the overhead of profiling with high accuracy and we hope the framework can also be implemented based on other dynamic instrumentation systems such as Pin[13] and Valgrind[15] etc. without too much additional work. In this paper, our framework is implemented based on DynamoRIO[5]. We adopt two-stage sampling strategy to instrument checking code and analysis code. The checking code periodically forks a process which is executed along with the original application to instrument and analyze the required information in parallel. Figure 1 presents the overview of our framework. After an application starting to execute, the monitor takes the control of the application immediately. Then it registers or loads the tool and initializes

the sampling strategy. The sampling module (the Sampler in Figure 1) drives the monitor to execute the checking and analysis code. The scheduler module monitors the overload of the whole system, schedules code slices among cores or CPUs. It also sends the system's load feedback to the sampling module, so that the sampling module can use the feedback to adjust the sampling rate. If a code slice terminates, its analysis data will be written to the shared memory or temporary file depending on the tool. When the application runs to the end, all the slices' data will be merged to construct the final result.

3.2 Two-Stage Sampling

It is well known that different programs executed on the same platform have different behavior. Therefore, it is not easy to determine when and where to execute instrumentation and analysis codes to perform a perfect profiling based on sampling. Shadow Profiling[14] uses the user specified parameters to determinate when to fork a shadow process directly from a signal handler function. However, leaving this burden to users is not perfect because users may not know how to specify these parameters. Our framework is different from it. We adopt two-stage sampling strategy so that we can utilize the runtime profiling information to determinate when and where to fork a code slice.

Figure 2 depicts our two-stage sampling strategy. On the first stage, we sample hardware performance counter to instrument the checking code (will be described in monitor module). In our framework, we use PAPI to initialize and to monitor hardware performance counter. We also provide a functional tool to specify a threshold and the event which needs to be monitored. When the hardware performance counter reaches the threshold, it will be overflowed and our overflow handler will be executed. In our overflow handler, we firstly check the load balance condition. If the condition is true, we set a flag and the next basic block (a single-entry, single-exit section of code with no internal control flow) will be instrumented by the checking code of our monitor. If the condition is false, we just skip this event. On second stage, the checking code examines the checking condition and decides whether to fork a code slice. If it forks a slice, the code slice will perform analysis and execute along with the application code in parallel.

3.3 The Monitor

The monitor takes the control of an application and registers the event handlers defined by the tool to DynamoRIO when an application starts executing under DynamoRIO. It also specifies the event and threshold, initializes the two-stage sampling strategy. As the application executes under DynamoRIO, the two-stage sampling event drives the monitor to implant checking code and analysis code. The main challenges of designing and implementing the monitor are two-fold: 1)implanting the checking code with low overhead;2)correctly handling system calls executed in code slices and multithread applications. The solutions for these challenges we proposed are presented in the following sections.

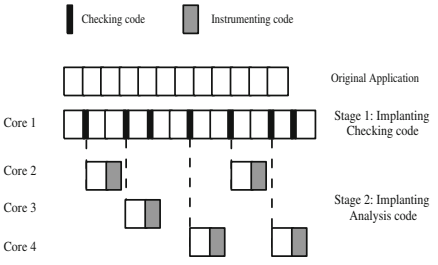


Fig. 2. The Two-stage sampling scheme

```
Instrumentation begin flags;

if (Cur_Tid != Instr_Tid ||Fork_Id !=Exe_Id)
{
    /*do nothing*/;
} else
{
    Save_Regs();
    Client_Fork();
    Restore_Regs();
}

Instrumentation end flags;
```

Fig. 3. Pseudo checking code

1)Instrumentation

In our framework, there are two tasks needed to be performed. At the first stage, we need to implant the checking code which is similar to the pseudo code shows in Figure 3. Since the scheduler provides the feedback information to the sampler, it can balance the sampling rate and system load at the first stage. So the checking work in our checking code is very simple. We associate each checking code with an *Instr_Tid* which is corresponding to the thread will execute the checking code and a *Fork_Id* respected the number of the checking code in the current thread. We also associate each thread with an *Exe_Id* respected the checking code it will execute. The checking code examines whether current thread id equals *Instr_Tid*, and whether *Fork_Id* equals *Exe_Id*. Using the *Fork_Id* and *Exe_Id*, we can avoid a checking point being executed many times in a loop. If both conditions are satisfied, we save registers which will be modified by *Client_Fork*. The *Client_Fork* call is our wrapper fork system call which is used to create code slice, to add slice information to a profile queue of management, and to restore saved registers. At the second stage, the code slice forked by the monitor will implant analysis code from tools. The monitor also emulates some system calls executed by the code slice.

A simple way to implant the checking code is to insert a clean call which is called by DynamoRIO after setting up the context. But it is often inefficient.

We adopt following methods to optimize the instrumenting code and reduce the overhead of the checking code. Firstly, we inline assembly instructions which do the same work as the pseudo code described before in the checking code. This reduces the number of instrumented instructions. Secondly, we perform register lifecycle analysis to identify registers that don't need to be saved and restored. Last but not least, we insert the instrumentation beginning and ending flags. When DynamoRIO builds traces, we can identify the checking code by detecting the instrumentation beginning and ending flags, and remove some checking codes that have been executed several times. Through this optimization, we reduce the execution times of the checking code since the trace is a hot code section which is frequently executed.

2)Muilthread

When a thread in multithreaded applications calls *fork()* to spawn a code slice, there will be only the calling thread survived in the code slice and other thread information is missed in the Linux kernel. There are two ways to solve this problem. One way is to implement a new system call which exactly copies all the thread information to the child like *forkall()* in Solaris 10 while it needs to modify the kernel and is non-portable. The other way is to recover the thread information in the user space, but it needs to recreate one kernel thread for each user thread and to synchronize those threads before the child begin to execute[14]. Since profiling is tolerant of some errors, these cumbersome methods are unnecessary.

Because each code slice's execution time is very short, our framework treat each thread individually and profiling them as a single thread. The monitor sets up the two-stage sampling strategy and allocates a thread describer to record the thread information, code slice list, and acquired lock list etc. for each application thread. When an application thread forks a code slice, the code slice is the same as the application thread while it only has one thread. Then the code slice is instrumented and executed along with application threads in parallel. Since its execution time is short, it is almost unnecessary to know other threads' execution. For some profiling focusing on synchronization between threads, it may not be suitable while it is acceptable for most profiling. Another problem for multithreaded applications is the deadlock. Since lock may be acquired by one thread while it is missed in the child when the child is forked, there will be a deadlock in the child. In our framework, we wrap pthread functions which acquire lock to record the address of the lock that acquired by the application thread and insert the address to a lock-acquired list. So in code slice we firstly release locks acquired by other application threads that are missed in the code slice.

3.4 The Scheduler

The scheduler has two main functions: 1) performing scheduling to achieve load balance and 2) adjusting the sampling rate according to the system overload information. Currently, we just create the code slice and let the Linux kernel schedule the code slices among different cores or CPUs. For the second function,

we define several variables to represent the system load. *CORE_NUM* presents the number of cores in the system. *SLICE_NUM* is the number of code slice. *CUR_INSTR_NUM* describes the number of checking codes that are implanted while are not executed yet. These variables are all dynamically updated. To achieve the best performance by exploiting thread parallelism, the thread number should not greater than the number of cores. So if the *SLICE_NUM* or *CUR_INSTR_NUM* is greater than *CORE_NUM* minus the number of the application thread, we skip the sampling. The scheduler also dynamically checks the execution state of code slices and confirms the termination of a code slice. If a code slice is handed up or executed over the time allocated for it, the scheduler kills the code slice by sending a SIGKILL signal.

3.5 Exported APIs

We also define APIs for tool developers to take the advantage of our two-stage sampling framework. We allow tool developers to register user-defined sampling events in our framework. This can guide whether or when we should fork a code slice and can change the second stage's default behavior. We also provide APIs to register the code slice begin and end events etc. Our experiments are performed by using these APIs.

Figure 4 presents the bbsize tool based on our framework. The tool is modified by using our APIs from the DynamoRIO *bbsize* tool. The tool firstly call mtp_init which accepts four arguments that specify the sampling event, threshold, maximum number of code slice, and the time when to initialize our framework. Then it registers the event callback function like normal DynamoRIO tools.

4 Performance Evaluation

In this section, we show the performance evaluation of our MT-Profiler.In particular, we evaluate and analyze the overhead of our optimization stragies used in the MT-Profiler.

4.1 Experiment Setup

In order to evaluate the performance of MT-Profiler, we use a machine with an Intel Xeon eight-core 1.6GHz processor with 4MB cache for each core and 4GB of main memory. The system is running Linux kernel version 2.6.18 patched with PerfCtr version 2.6.36 and PAPI 3.6. The benchmarks we used are from NPB3.3 OPENMP test suite, which are compiled with icc or ifort 11.0 using the -O3 flags. Each experiment presented in this paper is executed 10 times and takes the averages as results.

4.2 Performance Comparison

In our first experiment, we evaluate our framework by comparing the performance and accuracy with DynamoRIO performing full instrumentation. In this

```
#include "mtp.h"

static int num_bb;
static long total_size;

void count(int num)
 {
   total_size += num;
   num_bb++;
}

static void event_exit(void)
{
   char msg[512];
   int len;
   void *dr

   len = dr_snprintf(msg,sizeof(msg)/sizeof(msg[0]),
          "%d %d", num_bb, total_size);
   DR_ASSERT(len > 0);

   dr  = dr_get_current_drcontext();
   mtp_log_data(dr, "%s\n", msg);
}

static dr_emit_flags_t event_basic_block(void *drcontext, void *tag, instrlist_t *bb, bool for_trace, bool
translating)
{
   instr_t *instr, *cur, *next;
   int cur_size = 0;

   if (mtp_mode( ) == MTP_MODE_NORMAL) {
      return DR_EMIT_DEFAULT;
   }

   if (translating) {
        return DR_EMIT_DEFAULT;
   }

   cur = instrlist_first(bb);

   for (instr = cur; instr != NULL; instr = instr_get_next(instr)) {
         cur_size++;
   }

   dr_insert_clean_call(drcontext, bb, instrlist_first(bb), count, false, 1, OPND_CREATE_INT32(cur_size));
   return DR_EMIT_DEFAULT;
}

DR_EXPORT void  dr_init(client_id_t id)
{
     num_bb = 0;
     total_size = 0;

     mtp_init(MTP_TOT_CYC, 1000000, 0, 0);

     mtp_register_slide_exit_event(event_exit);
     mtp_register_bb_event(event_basic_block);
}
```

Fig. 4. The bbsize tool of MT-Profiler

experiment, we set each benchmark with two threads, choose the *PAPI_TOT_INS* as sampling event, and set sampling threshold to 5,000,000. We use the *bbsize* tool shown in the Figure 4 to count the average size of dynamic basic block against with the DynamoRIO *bbsize* tool which performs instrumentation at basic block level.

Figure 5 and Figure 6 shows the results of running the *bbsize* tool. We normalize the execution time of the DynamoRIO *bbsize* tool to our version of bbsize tool to calculate the speedup. Similarily, to calculate the accuracy, we use the dynamic basic block size obtained from DynamoRIO to divide the difference between the dynamic basic block sizes got from our framework and DynamoRIO.

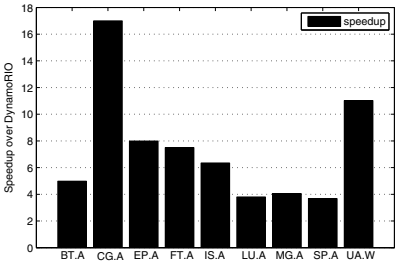


Fig. 5. The speedup of MT-Profiler over DynamoRIO

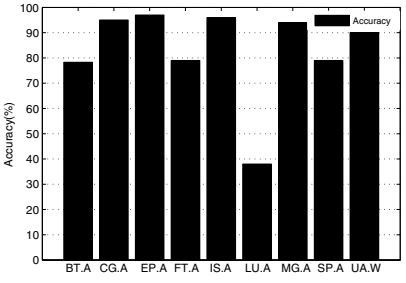


Fig. 6. The accuracy of MT-Profiler

As shown in Figure 5 and Figure 6, our version of *bbsize* tool achieves 2 to 17 times speedup over the traditional DynamoRIO *bbsize* tool with over 80% of accuracy on average. However, we can see from Figure 6 that lu.A's accuracy is lower than other benchmarks. This is because hot code sections are with large basic block size, leading the full instrumentation to get larger average size than the result of our version. But our experiment shows that higher accuracy can be obtained by adjusting the sampling threshold.

4.3 Runtime Overhead

In this experiment, we measure the runtime overhead of our framework. There are several factors that impact the runtime overhead: implanting and execution of the checking code, forking code slices to perform the profiling, and scheduling etc. We use the same experiment setting as that of the performance evaluation section. Figure 7 presents the overhead of each benchmark running on our framework. The native bar means the execution time of the benchmark which is executed without DynamoRIO, instrumentation and profiling. The instrumentation bar means the time of implanting and executing the checking code at the first stage. We normalize the overhead incurred by our framework to the native runs. As the result shows, the worst situation is two times slower than the native execution and on average the overhead is very low. We can easily conclude from the result that the overhead of instrumenting and executing the checking code

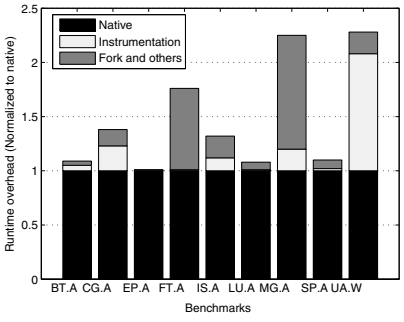


Fig. 7. The runtime overhead of MT-Profiler

is low with our optimization. However, we can observe that the fork & other overhead of ft.A and mg.A is higher than that of instrumentation and checking. We found that this overhead is incurred by executing the system call *sched_yield* to relinquish the processor, thus the main application's execution is slow down. This is mainly due to the active number of code slice is more than the number of hardware cores, and hence the operating system schedule the application thread to wait queues. The overhead can be reduced by adjusting the sample threshold and the maximum number of active code slice. Another exception is that the instrumentation and checking overhead of ua.W is higher than others. This is because the checking code is executed many times due to the loops.

4.4 Effects of Sampling Rate

In this section, we evaluate the effect of changing the sampling rate. In our first set of experiments, we tune sampling rate of the first stage to evaluate its effect. We configure the sampling threshold from 1M to 10M instructions which are executed completely. We chose mg.A because it has observed overhead incurred by each stage of our sampling strategy, so that it can best illustrate our experiment result. Figure 8 shows the result of changing the sampling rate of the first stage. We can see that the overhead of both stages decrease by increasing the sampling threshold. This is because with lower sampling rate the overhead caused by instrumenting and executing the checking code is decreased, so the overhead of the first stage is reduced. Furthermore, by reducing the number of checking codes, the overhead of the second stage is also reduced, since it reduces the number of forking points. In our second set of experiments, we change the sampling rate of the second stage by registering a sampling handler to change the default behavior through our exported APIs, while the sampling threshold of the first stage is configured as 5M instructions. We still chose mg.A to do our experiment. Figure 9 presents the results. We can observe that the overhead of the second stage is reduced by reducing the sampling rate of the second stage, while the overhead of the first stage is almost the same. We can also see that the fork & other overhead does not reduce to 50% when the sampling rate degrades

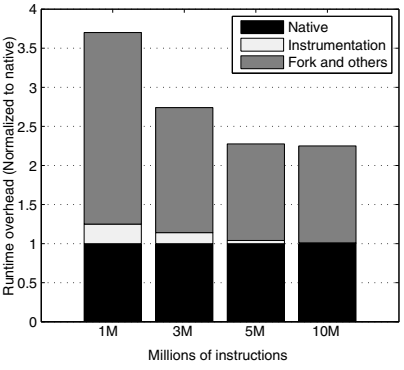


Fig. 8. The runtime overhead of MG.A on MT-Profiler when the first stage sampling rate varies

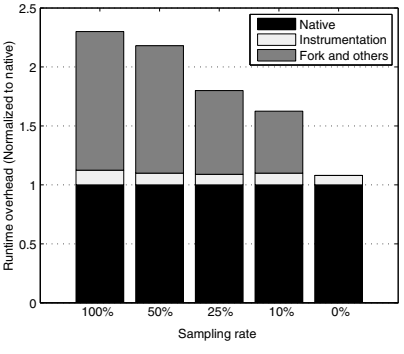


Fig. 9. The runtime overhead of MG.A on MT-Profiler when the second stage sampling rate varies

from 100% to 25%. This is because the scheduling overhead of the operating system is the main overhead, which we discussed before.

5 Conclusion

In this paper, we propose MT-Profiler which is a parallel dynamic profiling framework.It exploits hardware parallelism to reduce profiling overhead. Unlike prior approaches, MT-Profiler adopts two-stage sampling scheme to further speedup profiling. At the first stage, we sample the hardware performance counter to implant checking code. At the second stage, the checking code examines the checking condition to do further sampling work. If the checking point is hit, it forks a code slice which is analysis code and it executes along with the application thread in parallel. We also present our optimization strategies which are inlining assembly checking code, performing register lifecycle analysis, and trace optimizing to reduce the overhead of instrumenting and executing

checking codes. These methods can also be used in other dynamic analysis tools. Finally, we implemented our prototype system on DynamRIO. It can also be implemented easily in other dynamic instrumentation frameworks.

We design several experiments to evaluate our framework. Performance evaluation shows that our system obtains 2 to 17 times speedup while the accuracy is over 80% on average. The accuracy can also be tuned more highly by changing the sampling rate. We also evaluated the overhead of our framework and the sampling rate variation effect. The result shows that the average overhead of our framework is 148% slowdown relative to the native execution. We also show that our two-stage sampling scheme is tunable at both stages.

However, there still has room to improve the efficiency and usability of our framework. Currently, we just fork the code slice and let the operating system (OS) schedule threads. Our experiment shows that the main overhead of mg.A is caused by OS scheduling. We plan to perform dynamic system load monitoring and then schedule the code slice to idle cores to achieve higher efficiency. Secondly, our framework is not deterministic. Hence, each execution may have different profiling results for multi-threaded applications. In the future, We will handle this problem. Finally, the current version of MT-Profiler can only support DynamoRIO. We will port the current system to other instrumentation frameworks such as Pin[13]. to make more tools can benefit from our framework.

Acknowledgement. This work is supported by National 973 Basic Research Program of China under grant No.2007CB310900 and NSF China under grant No.60973036

References

1. Arnold, M., Ryder, B.: A framework for reducing the cost of instrumented code. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 168–179. ACM, New York (2001)
2. Arnold, M., Sweeney, P.: Approximating the calling context tree via sampling. Tech. rep., IBM T. J. Wastson Research Center (2000)
3. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: A transparent dynamic optimization system. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1–12. ACM, New York (2000)
4. Bond, M., McKinley, S.: Continuous Path and Edge Profiling. In: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2005), pp. 130–140. ACM, New York (2005)
5. Bruening, D.: Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Ph.D. thesis, MIT, Cambridge, Mass, USA (2004)
6. Cantrill, B., Shapiro, M., Leventhal, A.: Dynamic instrumentation of production systems. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation, pp. 15–28. USENIX (2004)
7. Chen, H., Hsu, W., Lu, J., Yew, P., Chen, D.: Dynamic trace selection using performance monitoring hardware sampling. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 79–90. IEEE, Los Alamitos (2003)

8. Dean, J., Hicks, J., Waldspurger, C., Weihl, W., Chrysos, G.: Proleme: Hardware support for instructionlevel proling on out-of-order processors. In: Proceedings of the International Symposium on Microarchitecture, pp. 292–302. IEEE, Los Alamitos (1997)
9. Ha, J., Arnold, M., Blackburn, S., McKinley, K.: A concurrent dynamic analysis framework for multicore hardware. In: Proceeding of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications, pp. 155–174. ACM, New York (2009)
10. Hirzel, M., Chilimbi, T.: Bursty tracing: A framework for low-overhead temporal proling. In: Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization, pp. 117–126. ACM, New York (2001)
11. Jennifer, A., Lance, B., Jeffrey, D., Sanjay, G., Monika, H., Shun-Tak, L., Richard, S., Mark, V., Carl, W., William, W.: Continuous profiling: Where have all the cycles gone?. ACM Transactions on Computer Systems (TOCS) 15(4), 357–390 (1997)
12. Larus, J.: Whole program paths. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 259–269. ACM, New York (1999)
13. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 191–200. ACM, New York (2005)
14. Moseley, T., Shye, A., Reddi, V., Grunwald, D., Peri, R.: Shadow proling: Hiding instrumentation costs with parallelism. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 198–208. IEEE, Los Alamitos (2007)
15. Nethercote, N., Seward, J.: Valgrind: A program supervision framework. Electronic Notes in Theoretical Computer Science 89(2), 1–23 (2003)
16. Patil, H., Fischer, C.: Efficient run-time monitoring using shadow processing. In: Proceedings of the Workshop on Automated and Algorithmic Debugging, pp. 119–132. Ghent University (1995)
17. Shetty, R., Kharbutli, M., Solihin, Y., Prvulovic, M.: HeapMon: a helper-thread approach to programmable, automatic, and low overhead memory bug detection. IBM Journal of Research and Development 50(2/3), 261–275 (2006)
18. Srivastava, A., Eustace, A.: ATOM: A system for building customized program analysis tools. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 196–205. ACM, New York (1994)
19. Wallace, S., Hazelwood, K.: Superpin: Parallelizing dynamic instrumentation for real-time performance. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 209–220. IEEE, Los Alamitos (2007)
20. Wang, C., Kim, H., Wu, Y., Ying, V.: Compiler-managed software-based redundant multi-threading for transient fault detection. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 244–258. IEEE, Los Alamitos (2007)
21. Zhao, Q., Cutcutache, I., Wong, W.: PiPA: Pipelined Proling and Analysis. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 185–194. IEEE, Los Alamitos (2008)
22. Zhou, P., Qin, F., Liu, W., Zhou, Y., Torrellas, J.: iWatcher: Efficient Architectural Support for Software Debugging. In: Proceedings of the ACM/IEEE International Symposium on Computer Architecture, pp. 224–235. ACM, New York (2004)

Author Index